The MUNIX- documentation is divided in 3 main parts, which are currently provided in 4 binders. The corresponding binder ist shown in bold face below.

MUNIX I	COMMANDS, PROGRAMMING, SPECIALS, MAINTENANCE
MUNIX I a (binder 1)	COMMANDS Introduction Release-Notes How to get started Table of contents Permuted Index Commands and Application Programms in alphabetic order
MUNIX I b (binder 2)	PROGRAMMING, SPECIALS, MAINTENANCE System calls Subroutines Special files File Formats Games Miscellaneous Facilities Maintenance
MUNIX II (binder 3)	TUTORIALS USER'S GUIDE PROGRAMMING GUIDE SUPPORT TOOLS GUIDE DOCUMENT PROCESSING GUIDE UNIX System V Init and Getty UUCP Tutorial UUCP Administrator's Manual UNIX System Accounting File System Checking LP Spooling System UNIX System Remote Job Entry

MUNIX
PROGRAMMING, SPECIALS, MAINTENANCE VOLUME I b

D930100V1be0484 Version 1.5/02

Information in this document is subject to change without notice and does not represent a commitment on the part of Periphere Computer Systeme GmbH. The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement.

Doc.-No.: D930100V1be0484

Trademarks: MUNIX for PCS
DEC, PDP for DEC
UNIX for Bell Laboratories

Copyright 1984 by
PCS GmbH, Pfaelzer-Wald-Strasse 36, D-8000 Muenchen 90, tel. 089/ 67804-0
The information contained herein is the property of PCS and shall neither be reproduced in whole or in part without PCS's prior written approval nor be implied to grant any licence to make, use, or sell equipment nanufactured in accordance herewith.

PCS reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented.

Copyright 1979, Bell Telephone Laboratories, Incorporated.
Holders of a UNIX^{TM} software license are permitted to copy this document, or any portion of it, as necessary for licensed use of the software, provided this copyright notice and statement of permission are included.

The MUNIX- documentation is divided in 3 main parts, which are currently provided in 4 binders. The corresponding binder ist shown in bold face below.

MUNIX I	COMMANDS, PROGRAMMING, SPECIALS, MAINTENANCE
MUNIX I a (binder 1)	COMMANDS Introduction Release-Notes How to get started Table of contents Permuted Index Commands and Application Programms in alphabetic order
MUNIX I b (binder 2)	**PROGRAMMING, SPECIALS, MAINTENANCE** **System calls** **Subroutines** **Special files** **File Formats** **Games** **Miscellaneous Facilities** **Maintenance**
MUNIX II (binder 3)	TUTORIALS USER'S GUIDE PROGRAMMING GUIDE SUPPORT TOOLS GUIDE DOCUMENT PROCESSING GUIDE UNIX System V Init and Getty UUCP Tutorial UUCP Administrator's Manual UNIX System Accounting File System Checking LP Spooling System UNIX System Remote Job Entry

MUNIX III	BASICS, OPTIONS
(binder 4)	Current MUNIX Commands - Summary
	Setting up MUNIX V1.5
	UNIX Programming
	Assembler Reference Manual
	Assembler 68000 User's Guide - outdated
	A Portable Fortran 77 Compiler
	A Tutorial Introduction to ADB
	CADMUS Testmonitor V1.0
	Bad Block Behandlung / Bad Sector Handling
	Typing Documents on the UNIX System
	Using the -MS Macros
	The UNIX Time-Sharing System
	UNIX Implementaion
	The UNIX I/O System
	Display Editing with VI
	Edit: A Tutorial
	Ex Reference Manual
Options:	MED, PASCAL
	Berkeley Extension Package

MUNIX I b

NAME

intro – introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always −1; the individual descriptions specify the details. An error number is also made available in the external variable *errno*. *Errno* is not cleared on successful calls, so it should be tested only after an error has been indicated.

All of the possible error numbers are not listed in each system call description because many errors are possible for most of the calls. The following is a complete list of the error numbers and their names as defined in <errno.h>.

1 EPERM Not owner
: Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 ENOENT No such file or directory
: This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3 ESRCH No such process
: No process can be found corresponding to that specified by *pid* in *kill* or *ptrace*.

4 EINTR Interrupted system call
: An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

5 EIO I/O error
: Some physical I/O error. This error may in some cases occur on a call following the one to which it actually applies.

6 ENXIO No such device or address
: I/O on a special file refers to a subdevice which does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not on-line or no disk pack is loaded on a drive.

7 E2BIG Arg list too long
: An argument list longer than 5,120 bytes is presented to a member of the *exec* family.

8 ENOEXEC Exec format error
: A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number (see *a.out*(5)).

DEFINITIONS

Process ID

Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 0 to 30,000.

Parent Process ID

A new process is created by a currently active process; see *fork*(2). The parent process ID of a process is the process ID of its creator.

Process Group ID

Each active process is a member of a process group that is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes; see *kill*(2).

Tty Group ID

Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to terminate a group of related process upon termination of one of the processes in the group; see *exit*(2) and *signal*(2).

Real User ID and Real Group ID

Each user allowed on the system is identified by a positive integer called a real user ID.

Each user is also a member of a group. The group is identified by a positive integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process.

Effective User ID and Effective Group ID

An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process's real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group ID bit set; see *exec*(2).

Super-user

A process is recognized as a ***super-user*** process and is granted special privileges if its effective user ID is 0.

Special Processes

The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to as *proc0* and *proc1*.

Proc0 is the scheduler. *Proc1* is the initialization process (*init*). Proc1 is the ancestor of every other process in the system and is used to control the process structure.

File Name.

Names consisting of 1 to 14 characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash).

Note that it is generally unwise to use *, ?, [, or] as part of file names because of the special meaning attached to these characters by the shell. See *sh*(1). Although permitted, it is advisable to avoid the use of unprintable characters in file names.

Path Name and Path Prefix

A path name is a null-terminated character string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a file name.

More precisely, a path name is a null-terminated character string constructed as follows:

```
<path-name>::=<file-name>|<path-prefix><file-name>|/
<path-prefix>::=<rtprefix>|/<rtprefix>
<rtprefix>::=<dirname>/|<rtprefix><dirname>/
```

where <file-name> is a string of 1 to 14 characters other than the ASCII slash and null, and <dirname> is a string of 1 to 14 characters (other than the ASCII slash and null) that names a directory.

If a path name begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory.

A slash by itself names the root directory.

Unless specifically stated otherwise, the null path name is treated as if it named a non-existent file.

Directory.

Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory.

Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. A process's root directory need not be the root directory of the root file system.

File Access Permissions.

Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.

The process's effective user ID does not match the user ID of the owner of the file, and the process's effective group ID matches the group of the file and the appropriate access bit of the "group" portion (070) of the file mode is set.

The process's effective user ID does not match the user ID of the owner of the file, and the process's effective group ID does not match the group ID of the file, and the appropriate access bit of the "other" portion (07) of the file mode is set.

Otherwise, the corresponding permissions are denied.

Message Queue Identifier

A message queue identifier (msqid) is a unique positive integer created by a *msgget*(2) system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```
struct ipc_perm msg_perm;/* operation permission struct */
ushort msg_qnum;          /* number of msgs on q */
ushort msg_qbytes;        /* max number of bytes on q */
ushort msg_lspid;         /* pid of last msgsnd operation */
ushort msg_lrpid;         /* pid of last msgrcv operation */
time_t msg_stime;         /* last msgsnd time */
time_t msg_rtime;         /* last msgrcv time */
time_t msg_ctime;         /* last change time */
                          /* Times measured in secs since */
                          /* 00:00:00 GMT, Jan. 1, 1970 */
```

Msg_perm is a ipc_perm structure that specifies the message operation permission (see below). This structure includes the following members:

```
ushort cuid;    /* creator user id */
ushort cgid;    /* creator group id */
ushort uid;     /* user id */
ushort gid;     /* group id */
ushort mode;    /* r/w permission */
```

Msg_qnum is the number of messages currently on the queue. **Msg_qbytes** is the maximum number of bytes allowed on the queue. **Msg_lspid** is the process id of the last process that performed a *msgsnd* operation. **Msg_lrpid** is the process id of the last process that performed a *msgrcv* operation. **Msg_stime** is the time of the last *msgsnd* operation, **msg_rtime** is the time of the last *msgrcv* operation, and **msg_ctime** is the time of the last *msgctl*(2) operation that changed a member of the above structure.

Message Operation Permissions.

In the *msgop*(2) and *msgctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Write by user
00060	Read, Write by group
00006	Read, Write by others

Read and Write permissions on a msqid are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches **msg_perm.[c]uid** in the data structure associated with *msqid* and the appropriate bit of the "user" portion (0600) of **msg_perm.mode** is set.

The process's effective user ID does not match **msg_perm.[c]uid** and the process's effective group ID matches **msg_perm.[c]gid** and the appropriate bit of the "group" portion (060) of **msg_perm.mode** is set.

The process's effective user ID does not match **msg_perm.[c]uid** and the process's effective group ID does not match **msg_perm.[c]gid** and the appropriate bit of the "other" portion (06) of **msg_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Semaphore Identifier

A semaphore identifier (semid) is a unique positive integer created by a *semget*(2) system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

```
struct ipc_perm sem_perm;/* operation permission struct */
ushort sem_nsems;         /* number of sems in set */
time_t sem_otime;         /* last operation time */
time_t sem_ctime;         /* last change time */
                          /* Times measured in secs since */
                          /* 00:00:00 GMT, Jan. 1, 1970 */
```

Sem_perm is a ipc_perm structure that specifies the semaphore operation permission (see below). This structure includes the following members:

```
ushort cuid;        /* creator user id */
ushort cgid;        /* creator group id */
ushort uid;         /* user id */
ushort gid;         /* group id */
ushort mode;        /* r/a permission */
```

The value of **sem_nsems** is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. Sem_num values run sequentially from 0 to the value of sem_nsems minus 1. **Sem_otime** is the time of the last *semop*(2) operation, and **sem_ctime** is the time of the last *semctl*(2) operation that changed a member of the above structure.

A semaphore is a data structure that contains the following members:

```
ushort semval;    /* semaphore value */
short  sempid;    /* pid of last operation  */
ushort semncnt;   /* # awaiting semval > cval */
ushort semzcnt;   /* # awaiting semval = 0 */
```

Semval is a non-negative integer. **Sempid** is equal to the process ID of the last process that performed a semaphore operation on this semaphore. **Semncnt** is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become greater than its current value. **Semzcnt** is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.

Semaphore Operation Permissions.

In the *semop*(2) and *semctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Alter by user
00060	Read, Alter by group
00006	Read, Alter by others

Read and Alter permissions on a semid are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches **sem_perm.[c]uid** in the data structure associated with *semid* and the appropriate bit of the "user" portion (0600) of **sem_perm.mode** is set.

The process's effective user ID does not match **sem_perm.[c]uid** and the process's effective group ID matches **sem_perm.[c]gid** and the appropriate bit of the "group" portion (060) of **sem_perm.mode** is set.

The process's effective user ID does not match **sem_perm.[c]uid** and the process's effective group ID does not match **sem_perm.[c]gid** and the appropriate bit of the "other" portion (06) of **sem_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Shared Memory Identifier

A shared memory identifier (shmid) is a unique positive integer created by a *shmget*(2) system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. The data structure is referred to as *shmid_ds* and contains the following members:

```
struct ipc_perm shm_perm;/* operation permission struct */
int    shm_segsz;         /* size of segment */
ushort shm_cpid;          /* creator pid */
ushort shm_lpid;          /* pid of last operation */
short  shm_nattch;        /* number of current attaches */
time_t shm_atime;         /* last attach time */
time_t shm_dtime;         /* last detach time */
time_t shm_ctime;         /* last change time */
                          /* Times measured in secs since */
                          /* 00:00:00 GMT, Jan. 1, 1970 */
```

Shm_perm is a ipc_perm structure that specifies the shared memory operation permission (see below). This structure includes the following members:

```
ushort cuid;        /* creator user id */
ushort cgid;        /* creator group id */
ushort uid;         /* user id */
ushort gid;         /* group id */
ushort mode;        /* r/w permission */
```

Shm_segsz specifies the size of the shared memory segment. Shm_cpid is the process id of the process that created the shared memory identifier. Shm_lpid is the process id of the last process that performed a *shmop*(2) operation. Shm_nattch is the number of processes that currently have this segment attached. **Shm_atime** is the time of the last *shmat* operation, **shm_dtime** is the time of the last *shmdt* operation, and **shm_ctime** is the time of the last *shmctl*(2) operation that changed one of the members of the above structure.

Shared Memory Operation Permissions.

In the *shmop*(2) and *shmctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Write by user
00060	Read, Write by group
00006	Read, Write by others

Read and Write permissions on a shmid are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches **shm_perm.[c]uid** in the data structure associated with *shmid* and the appropriate bit of the "user" portion (0600) of **shm_perm.mode** is set.

The process's effective user ID does not match **shm_perm.[c]uid** and the process's effective group ID matches **shm_perm.[c]gid** and the appropriate bit of the "group" portion (060) of **shm_perm.mode** is set.

The process's effective user ID does not match **shm_perm.[c]uid** and the process's effective group ID does not match **shm_perm.[c]gid** and the appropriate bit of the "other" portion (06) of **shm_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

SEE ALSO

intro(3).

NAME

access – determine accessibility of a file

SYNOPSIS

int access (path, amode)
char *path;
int amode;

DESCRIPTION

Path points to a path name naming a file. *Access* checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. The bit pattern contained in *amode* is constructed as follows:

04	read
02	write
01	execute (search)
00	check existence of file

Access to the file is denied if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

Read, write, or execute (search) permission is requested for a null path name. [ENOENT]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

Write access is requested for a file on a read-only file system. [EROFS]

Write access is requested for a pure procedure (shared text) file that is being executed. [ETXTBSY]

Permission bits of the file mode do not permit the requested access. [EACCES]

Path points outside the process's allocated address space. [EFAULT]

The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits, members of the file's group other than the owner have permissions checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.

RETURN VALUE

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), stat(2).

NAME

acct – enable or disable process accounting

SYNOPSIS

int acct (path)
char *path;

DESCRIPTION

Acct is used to enable or disable the system's process accounting routine. If the routine is enabled, an accounting record will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal; see *exit*(2) and *signal*(2). The effective user ID of the calling process must be super-user to use this call.

Path points to a path name naming the accounting file. The accounting file format is given in *acct*(5).

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

Acct will fail if one or more of the following are true:

The effective user ID of the calling process is not super-user. [EPERM]

An attempt is being made to enable accounting when it is already enabled. [EBUSY]

A component of the path prefix is not a directory. [ENOTDIR]

One or more components of the accounting file's path name do not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

The file named by *path* is not an ordinary file. [EACCES]

Mode permission is denied for the named accounting file. [EACCES]

The named file is a directory. [EISDIR]

The named file resides on a read-only file system. [EROFS]

Path points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

acct(5).

NAME

alarm – set a process's alarm clock

SYNOPSIS

unsigned alarm (sec)
unsigned sec;

DESCRIPTION

Alarm instructs the calling process's alarm clock to send the signal **SIGALRM** to the calling process after the number of real time seconds specified by *sec* have elapsed; see *signal*(2).

Alarm requests are not stacked; successive calls reset the calling process's alarm clock.

If *sec* is 0, any previously made alarm request is canceled.

RETURN VALUE

Alarm returns the amount of time previously remaining in the calling process's alarm clock.

SEE ALSO

pause(2), signal(2).

NAME

brk, sbrk – change data segment space allocation

SYNOPSIS

```
int brk (endds)
char *endds;

char *sbrk (incr)
int incr;
```

DESCRIPTION

Brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment; see *exec*(2). The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. The newly allocated space is set to zero.

Brk sets the break value to ***endds*** and changes the allocated space accordingly.

Sbrk adds *incr* bytes to the break value and changes the allocated space accordingly. *Incr* can be negative, in which case the amount of allocated space is decreased.

Brk and *sbrk* will fail without making any change in the allocated space if one or more of the following are true:

> Such a change would result in more space being allocated than is allowed by a system-imposed maximum (see *ulimit*(2)). [ENOMEM]
>
> Such a change would result in the break value being greater than or equal to the start address of any attached shared memory segment (see *shmop*(2)).

RETURN VALUE

Upon successful completion, ***brk*** returns a value of 0 and *sbrk* returns the old break value. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2).

NAME

chdir – change working directory

SYNOPSIS

```
int chdir (path)
char *path;
```

DESCRIPTION

Path points to the path name of a directory. *Chdir* causes the named directory to become the current working directory, the starting point for path searches for path names not beginning with /.

Chdir will fail and the current working directory will be unchanged if one or more of the following are true:

A component of the path name is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

Search permission is denied for any component of the path name. [EACCES]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

chroot(2).

NAME

chmod – change mode of file

SYNOPSIS

int chmod (path, mode)
char *path;
int mode;

DESCRIPTION

Path points to a path name naming a file. *Chmod* sets the access permission portion of the named file's mode according to the bit pattern contained in *mode*.

Access permission bits are interpreted as follows:

04000	Set user ID on execution.
02000	Set group ID on execution.
01000	Save text image after execution
00400	Read by owner
00200	Write by owner
00100	Execute (or search if a directory) by owner
00070	Read, write, execute (search) by group
00007	Read, write, execute (search) by others

The effective user ID of the process must match the owner of the file or be super-user to change the mode of a file.

If the effective user ID of the process is not super-user, mode bit 01000 (save text image on execution) is cleared.

If the effective user ID of the process is not super-user or the effective group ID of the process does not match the group ID of the file, mode bit 02000 (set group ID on execution) is cleared.

If an executable file is prepared for sharing then mode bit 01000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Thus, when the next user of the file executes it, the text need not be read from the file system but can simply be swapped in, saving time.

Chmod will fail and the file mode will be unchanged if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The effective user ID does not match the owner of the file and the effective user ID is not super-user. [EPERM]

The named file resides on a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

chown(2), mknod(2).

NAME

chown – change owner and group of a file

SYNOPSIS

```
int chown (path, owner, group)
char *path;
int owner, group;
```

DESCRIPTION

Path points to a path name naming a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

Only processes with effective user ID equal to the file owner or super-user may change the ownership of a file.

If *chown* is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

Chown will fail and the owner and group of the named file will remain unchanged if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The effective user ID does not match the owner of the file and the effective user ID is not super-user. [EPERM]

The named file resides on a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2).

NAME

chroot – change root directory

SYNOPSIS

int chroot (path)
char *path;

DESCRIPTION

Path points to a path name naming a directory. *Chroot* causes the named directory to become the root directory, the starting point for path searches for path names beginning with **/**.

The effective user ID of the process must be super-user to change the root directory.

Whereas standard Unix always interprets /.. to refer to the same directory as /, **MUNIX** really goes one directory up for /.., i.e. after a chroot(/bin/usr) /.. will be the same as /bin before the chroot. This has been done to allow a virtual superroot for the Newcastle Connection.

Chroot will fail and the root directory will remain unchanged if one or more of the following are true:

Any component of the path name is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

The effective user ID is not super-user. [EPERM]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

chdir(2).

NAME
close – close a file descriptor

SYNOPSIS
int close (fildes)
int fildes;

DESCRIPTION
Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Close* closes the file descriptor indicated by *fildes*.

Close will fail if *fildes* is not a valid open file descriptor. [EBADF]

RETURN VALUE
Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO
creat(2), dup(2), exec(2), fcntl(2), open(2), pipe(2).

NAME

creat – create a new file or rewrite an existing one

SYNOPSIS

```
int creat (path, mode)
char *path;
int mode;
```

DESCRIPTION

Creat creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *path*.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the process's effective user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows:

> All bits set in the process's file mode creation mask are cleared. See *umask*(2).
>
> The "save text image after execution bit" of the mode is cleared. See *chmod*(2).

Upon successful completion, a non-negative integer, namely the file descriptor, is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across *exec* system calls. See *fcntl*(2). No process may have more than 20 files open simultaneously. A new file may be created with a mode that forbids writing.

Creat will fail if one or more of the following are true:

> A component of the path prefix is not a directory. [ENOTDIR]
>
> A component of the path prefix does not exist. [ENOENT]
>
> Search permission is denied on a component of the path prefix. [EACCES]
>
> The path name is null. [ENOENT]
>
> The file does not exist and the directory in which the file is to be created does not permit writing. [EACCES]
>
> The named file resides or would reside on a read-only file system. [EROFS]
>
> The file is a pure procedure (shared text) file that is being executed. [ETXTBSY]
>
> The file exists and write permission is denied. [EACCES]
>
> The named file is an existing directory. [EISDIR]
>
> Twenty (20) file descriptors are currently open. [EMFILE]
>
> *Path* points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

close(2), dup(2), lseek(2), open(2), read(2), umask(2), write(2).

NAME

dup – duplicate an open file descriptor

SYNOPSIS

int dup (fildes)
int fildes;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Dup* returns a new file descriptor having the following in common with the original:

Same open file (or pipe).

Same file pointer. (i.e., both file descriptors share one file pointer.)

Same access mode (read, write or read/write).

The new file descriptor is set to remain open across *exec* system calls. See *fcntl*(2).

The file descriptor returned is the lowest one available.

Dup will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Twenty (20) file descriptors are currently open. [EMFILE]

RETURN VALUE

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), close(2), exec(2), fcntl(2), open(2), pipe(2).

NAME

execl, execv, execle, execve, execlp, execvp – execute a file

SYNOPSIS

```
int execl (path, arg0, arg1, ..., argn, (char *)0)
char *path, *arg0, *arg1, ..., *argn;

int execv (path, argv)
char *path, *argv[ ];

int execle (path, arg0, arg1, ..., argn, (char *)0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[ ];

int execve (path, argv, envp)
char *path, *argv[ ], *envp[ ];

int execlp (file, arg0, arg1, ..., argn, (char *)0)
char *file, *arg0, *arg1, ..., *argn;

int execvp (file, argv)
char *file, *argv[ ];
```

DESCRIPTION

Exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the *new process file*. This file consists of a header (see *a.out*(5)), a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss). There can be no return from a successful *exec* because the calling process is overlaid by the new process.

When a C program is executed, it is called as follows:

```
main (argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

Path points to a path name that identifies the new process file.

File points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH =" (see *environ*(7)). The environment is supplied by the shell (see *sh*(1)).

Arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

Argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). *Argv* is terminated by a null pointer.

Envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. *Envp* is terminated by a null pointer. For *execl* and *execv*, the C run-time start-off routine places a pointer to the calling process's environment in the global cell:

extern char **environ;

and it is used to pass the calling process's environment to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl*(2). For those file descriptors that remain open, the file pointer is unchanged.

Signals set to terminate the calling process will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to terminate new process; see *signal*(2).

If the set-user-ID mode bit of the new process file is set (see *chmod*(2)), *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process.

The shared memory segments attached to the calling process will not be attached to the new process (see *shmop*(2)).

Profiling is disabled for the new process; see *profil*(2).

The new process also inherits the following attributes from the calling process:

nice value (see *nice*(2))
process ID
parent process ID
process group ID
semadj values, when implemented (see *semop*(2))
tty group ID (see *exit*(2) and *signal*(2))
trace flag (see *ptrace*(2) request 0)
time left until an alarm clock signal (see *alarm*(2))
current working directory
root directory
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))
utime, *stime*, *cutime*, and *cstime* (see *times*(2))

Exec will fail and return to the calling process if one or more of the following are true:

One or more components of the new process file's path name do not exist. [ENOENT]

A component of the new process file's path prefix is not a directory. [ENOTDIR]

Search permission is denied for a directory listed in the new process file's path prefix. [EACCES]

The new process file is not an ordinary file. [EACCES]

The new process file mode denies execution permission. [EACCES]

The exec is not an *execlp* or *execvp*, and the new process file has the appropriate access permission but an invalid magic number in its header. [ENOEXEC]

The new process file is a pure procedure (shared text) file that is currently open for writing by some process. [ETXTBSY]

The new process requires more memory than is allowed by the system-imposed maximum MAXMEM. [ENOMEM]

The number of bytes in the new process's argument list is greater than the system-imposed limit of 5120 bytes. [E2BIG]

The new process file is not as long as indicated by the size values in its header. [EFAULT]

Path, *argv*, or *envp* point to an illegal address. [EFAULT]

RETURN VALUE

If *exec* returns to the calling process an error has occurred; the return value will be −1 and *errno* will be set to indicate the error.

SEE ALSO

exit(2), fork(2), environ(7).

NAME

exit, _exit – terminate process

SYNOPSIS

void exit (status)
int status;
void _exit (status)
int status;

DESCRIPTION

Exit terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination and the low order eight bits (i.e., bits 0377) of *status* are made available to it; see *wait*(2).

If the parent process of the calling process is not executing a *wait*, the calling process is transformed into a zombie process. A *zombie process* is a process that only occupies a slot in the process table, it has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see **<sys/proc.h>**) to be used by *times*.

The parent process ID of all of the calling process's existing child processes and zombie processes is set to 1. This means the initialization process (see *intro*(2)) inherits each of these processes.

Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.

For each semaphore for which the calling process has set a semadj value (see *semop*(2)), that semadj value is added to the semval of the specified semaphore.

If the process has a process, text, or data lock, an *unlock* is performed (see *plock*(2)).

An accounting record is written on the accounting file if the system's accounting routine is enabled; see *acct*(2).

If the process ID, tty group ID, and process group ID of the calling process are equal, the SIGHUP signal is sent to each processes that has a process group ID equal to that of the calling process.

The C function *exit* may cause cleanup actions before the process exits. The function *_exit* circumvents all cleanup.

SEE ALSO

signal(2), wait(2).

WARNING

See *WARNING* in *signal*(2).

NAME

fcntl – file control

SYNOPSIS

#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd, arg;

DESCRIPTION

Fcntl provides for control over open files. *Fildes* is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

The *cmds* available are:

F_DUPFD
: Return a new file descriptor as follows:

  Lowest numbered available file descriptor greater than or equal to *arg*.

  Same open file (or pipe) as the original file.

  Same file pointer as the original file (i.e., both file descriptors share one file pointer).

  Same access mode (read, write or read/write).

  Same file status flags (i.e., both file descriptors share the same file status flags).

  The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(2) system calls.

F_GETFD
: Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is 0 the file will remain open across *exec*, otherwise the file will be closed upon execution of *exec*.

F_SETFD
: Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg* (0 or 1 as above).

F_GETFL
: Get *file* status flags.

F_SETFL
: Set *file* status flags to *arg*. Only certain flags can be set; see *fcntl*(7).

Fcntl will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Cmd is F_DUPFD and 20 file descriptors are currently open. [EMFILE]

Cmd is F_DUPFD and *arg* is negative or greater than 20. [EINVAL]

RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD
: A new file descriptor.

F_GETFD
: Value of flag (only the low-order bit is defined).

F_SETFD
: Value other than −1.

F_GETFL
: Value of file flags.

F_SETFL
: Value other than −1.

Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

close(2), exec(2), open(2), fcntl(7).

NAME

fork – create a new process

SYNOPSIS

int fork ()

DESCRIPTION

Fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

environment
close-on-exec flag (see *exec*(2))
signal handling settings (i.e., SIG_DFL, SIG_IGN, function address)
set-user-ID mode bit
set-group-ID mode bit
profiling on/off status
nice value (see *nice*(2))
all attached shared memory segments, when implemented (see *shmop*(2))
process group ID
tty group ID (see *exit*(2) and *signal*(2))
trace flag (see *ptrace*(2) request 0)
time left until an alarm clock signal (see *alarm*(2))
current working directory
root directory
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))

The child process differs from the parent process in the following ways:

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

When implemented, all semadj values are cleared (see *semop*(2)).

Process locks, text locks and data locks are not inherited by the child (see *plock*(2)).

The child process's *utime*, *stime*, *cutime*, and *cstime* are set to 0.

Fork will fail and no child process will be created if one or more of the following are true:

The system-imposed limit on the total number of processes under execution would be exceeded. [EAGAIN]

The system-imposed limit on the total number of processes under execution by a single user would be exceeded. [EAGAIN]

RETURN VALUE

Upon successful completion, *fork* returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of –1 is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

SEE ALSO

exec(2), times(2), wait(2).

NAME

getpid, getpgrp, getppid – get process, process group, and parent process IDs

SYNOPSIS

int getpid ()

int getpgrp ()

int getppid ()

DESCRIPTION

Getpid returns the process ID of the calling process.

Getpgrp returns the process group ID of the calling process.

Getppid returns the parent process ID of the calling process.

SEE ALSO

exec(2), fork(2), intro(2), setpgrp(2), signal(2).

NAME

getuid, geteuid, getgid, getegid – get real user, effective user, real group, and effective group IDs

SYNOPSIS

int getuid ()

int geteuid ()

int getgid ()

int getegid ()

DESCRIPTION

Getuid returns the real user ID of the calling process.

Geteuid returns the effective user ID of the calling process.

Getgid returns the real group ID of the calling process.

Getegid returns the effective group ID of the calling process.

SEE ALSO

intro(2), setuid(2).

NAME

hertz – get the line frequency on the current machine

SYNOPSIS

int hertz ()

DESCRIPTION

Hertz returns either 50 or 60, depending on the line frequency. The system call returns the value of the definition of HERTZ in */usr/sys/conf.h*, which must have been set up properly at system generation time.

CAUTION

This system call is nonstandard. It is however necessary if programs like *time(1)* must give identical results on both sides of the Atlantic.

NAME

ioctl – control device

SYNOPSIS

ioctl (fildes, request, arg)
char *arg;

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). The writeups of various devices in Section 7 discuss how *ioctl* applies to them.

Ioctl will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Fildes is not associated with a character special device. [ENOTTY]

Request or *arg* is not valid. See Section 7. [EINVAL]

RETURN VALUE

If an error has occurred, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

termio(4)

NAME

kill – send a signal to a process or a group of processes

SYNOPSIS

int kill (pid, sig)
int pid, sig;

DESCRIPTION

Kill sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in *signal*(2), or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process unless, the effective user ID of the sending process is super-user.

The processes with a process ID of 0 and a process ID of 1 are special processes (see *intro*(2)) and will be referred to below as *proc0* and *proc1* respectively.

If *pid* is greater than zero, *sig* will be sent to the process whose process ID is equal to *pid*. *Pid* may equal 1.

If *pid* is 0, *sig* will be sent to all processes excluding *proc0* and *proc1* whose process group ID is equal to the process group ID of the sender.

If *pid* is −1 and the effective user ID of the sender is not super-user, *sig* will be sent to all processes excluding *proc0* and *proc1* whose real user ID is equal to the effective user ID of the sender.

If *pid* is −1 and the effective user ID of the sender is super-user, *sig* will be sent to all processes excluding *proc0* and *proc1*.

If *pid* is negative but not −1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

Kill will fail and no signal will be sent if one or more of the following are true:

Sig is not a valid signal number. [EINVAL]

No process can be found corresponding to that specified by *pid*. [ESRCH]

The user ID of the sending process is not super-user, and its real or effective user ID does not match the real or effective user ID of the receiving process. [EPERM]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

kill(1), getpid(2), setpgrp(2), signal(2).

NAME

link – link to a file

SYNOPSIS

int link (path1, path2)
char *path1, *path2;

DESCRIPTION

Path1 points to a path name naming an existing file. *Path2* points to a path name naming the new directory entry to be created. *Link* creates a new link (directory entry) for the existing file.

Link will fail and no link will be created if one or more of the following are true:

A component of either path prefix is not a directory. [ENOTDIR]

A component of either path prefix does not exist. [ENOENT]

A component of either path prefix denies search permission. [EACCES]

The file named by *path1* does not exist. [ENOENT]

The link named by *path2* exists. [EEXIST]

The file named by *path1* is a directory and the effective user ID is not super-user. [EPERM]

The link named by *path2* and the file named by *path1* are on different logical devices (file systems). [EXDEV]

Path2 points to a null path name. [ENOENT]

The requested link requires writing in a directory with a mode that denies write permission. [EACCES]

The requested link requires writing in a directory on a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

unlink(2).

NAME

long – system calls modified for long arguments

SYNOPSIS

```
long lread (fildes, buf, nbyte)
int fildes;
char *buf;
long nbyte;

long lwrite (fildes, buf, nbyte)
int fildes;
char *buf;
long nbyte;

char *lsbrk (incr)
long incr;
```

DESCRIPTION

These system calls are the same as their "non-l" counterparts except that they have a long instead of an int argument. They are available only in the two byte integer standard library.

NAME

lseek – move read/write file pointer

SYNOPSIS

long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;

DESCRIPTION

Fildes is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call. *Lseek* sets the file pointer associated with *fildes* as follows:

If *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location as measured in bytes from the beginning of the file is returned.

Lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

Fildes is not an open file descriptor. [EBADF]

Fildes is associated with a pipe or fifo. [ESPIPE]

Whence is not 0, 1 or 2. [EINVAL and SIGSYS signal]

The resulting file pointer would be negative. [EINVAL]

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

RETURN VALUE

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2).

NAME

mknod – make a directory, or a special or ordinary file

SYNOPSIS

int mknod (path, mode, dev)
char *path;
int mode, dev;

DESCRIPTION

Mknod creates a new file named by the path name pointed to by *path*. The mode of the new file is initialized from *mode*. Where the value of *mode* is interpreted as follows:

```
0170000 file type; one of the following:
        0010000 fifo special
        0020000 character special
        0040000 directory
        0060000 block special
        0100000 or 0000000 ordinary file
0004000 set user ID on execution
0002000 set group ID on execution
0001000 save text image after execution
0000777 access permissions; constructed from the following
        0000400 read by owner
        0000200 write by owner
        0000100 execute (search on directory) by owner
        0000070 read, write, execute (search) by group
        0000007 read, write, execute (search) by others
```

The file's owner ID is set to the process's effective user ID. The file's group ID is set to the process's effective group ID.

Values of *mode* other than those above are undefined and should not be used. The low-order 9 bits of *mode* are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared. See *umask*(2). If *mode* indicates a block or character special file, *dev* is a configuration dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

Mknod may be invoked only by the super-user for file types other than FIFO special.

Mknod will fail and the new file will not be created if one or more of the following are true:

The process's effective user ID is not super-user. [EPERM]

A component of the path prefix is not a directory. [ENOTDIR]

A component of the path prefix does not exist. [ENOENT]

The directory in which the file is to be created is located on a read-only file system. [EROFS]

The named file exists. [EEXIST]

Path points outside the process's allocated address space. [EFAULT]

NAME

mount – mount a file system

SYNOPSIS

int mount (spec, dir, rwflag)
char *spec, *dir;
int rwflag;

DESCRIPTION

Mount requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *Spec* and *dir* are pointers to path names.

Upon successful completion, references to the file *dir* will refer to the root directory on the mounted file system.

The low-order bit of *rwflag* is used to control write permission on the mounted file system; if 1, writing is forbidden, otherwise writing is permitted according to individual file accessibility.

Mount may be invoked only by the super-user.

Mount will fail if one or more of the following are true:

The effective user ID is not super-user. [EPERM]

Any of the named files does not exist. [ENOENT]

A component of a path prefix is not a directory. [ENOTDIR]

Spec is not a block special device. [ENOTBLK]

The device associated with *spec* does not exist. [ENXIO]

Dir is not a directory. [ENOTDIR]

Spec or *dir* points outside the process's allocated address space. [EFAULT]

Dir is currently mounted on, is someone's current working directory or is otherwise busy. [EBUSY]

The device associated with *spec* is currently mounted. [EBUSY]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

umount(2).

NAME

nice – change priority of a process

SYNOPSIS

```
int nice (incr)
int incr;
```

DESCRIPTION

Nice adds the value of *incr* to the nice value of the calling process. A process's *nice value* is a positive number for which a more positive value results in lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. Requests for values above or below these limits result in the nice value being set to the corresponding limit.

Nice will fail and not change the nice value if *incr* is negative and the effective user ID of the calling process is not super-user. [EPERM]

RETURN VALUE

Upon successful completion, *nice* returns the new nice value minus 20. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

nice(1), exec(2).

NAME

open – open for reading or writing

SYNOPSIS

#include <fcntl.h>
int open (path, oflag [, mode])
char *path;
int oflag, mode;

DESCRIPTION

Path points to a path name naming a file. *Open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. *Oflag* values are constructed by or-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY
Open for reading only.

O_WRONLY
Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY
This flag may affect subsequent reads and writes. See *read*(2) and *write*(2).

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

An *open* for reading-only will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

If O_NDELAY is clear:

The open will block until carrier is present.

O_APPEND
If set, the file pointer will be set to the end of the file prior to each write.

O_CREAT If the file exists, this flag has no effect. Otherwise, the file's owner ID is set to the process's effective user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows (see *creat*(2)):

All bits set in the process's file mode creation mask are cleared. See *umask*(2).

The "save text image after execution bit" of the mode is cleared. See *chmod*(2).

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL If O_EXCL and O_CREAT are set, *open* will fail if the file exists.

Upon successful completion a non-negative integer, the file descriptor, is returned.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *exec* system calls. See *fcntl*(2).

No process may have more than 20 file descriptors open simultaneously.

The named file is opened unless one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

O_CREAT is not set and the named file does not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

Oflag permission is denied for the named file. [EACCES]

The named file is a directory and *oflag* is write or read/write. [EISDIR]

The named file resides on a read-only file system and *oflag* is write or read/write. [EROFS]

Twenty (20) file descriptors are currently open. [EMFILE]

The named file is a character special or block special file, and the device associated with this special file does not exist. [ENXIO]

The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write. [ETXTBSY]

Path points outside the process's allocated address space. [EFAULT]

O_CREAT and O_EXCL are set, and the named file exists. [EEXIST]

O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading. [ENXIO]

RETURN VALUE

Upon successful completion, a non-negative integer, namely a file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

close(2), creat(2), dup(2), fcntl(2), lseek(2), read(2), write(2).

NAME

pause – suspend process until signal

SYNOPSIS

pause ()

DESCRIPTION

Pause suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

If the signal causes termination of the calling process, *pause* will not return.

If the signal is *caught* by the calling process and control is returned from the signal catching-function (see *signal*(2)), the calling process resumes execution from the point of suspension; with a return value of −1 from *pause* and *errno* set to EINTR.

SEE ALSO

alarm(2), kill(2), signal(2), wait(2).

NAME

phys – map physical memory

SYNOPSIS

```
char * phys(physaddr,size)
char *physaddr;
long size;
```

DESCRIPTION

Physaddr is a physical address greater than TOPMEM (see /usr/sys/conf.h, *newconf*(8)) and less than 0x400000. *Size* is a long integer greater than 0 and less than 0x100000. *Phys* returns a logical address *logaddr*. The physical address range from *physaddr* to *physaddr+size* is mapped into the logical address range from *logaddr* to *logaddr+size*. At most three phys calls may be made in a program. You can reset the foregoing phys calls with the call phys(0L,0L).

RETURN VALUE

Upon successful completion a logical address is returned that is mapped to the given physical address. Otherwise, a −1 is returned and *errno* is set to EINVAL.

NAME

pipe – create an interprocess channel

SYNOPSIS

int pipe (fildes)
int fildes[2];

DESCRIPTION

Pipe creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. *Fildes*[0] is opened for reading and *fildes*[1] is opened for writing.

Writes up to 5120 bytes of data are buffered by the pipe before the writing process is blocked. A read on file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out basis.

No process may have more than 20 file descriptors open simultaneously.

Pipe will fail if 19 or more file descriptors are currently open. [EMFILE]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

sh(1), read(2), write(2).

NAME

plock – lock process, text, or data in memory

SYNOPSIS

#include <sys/lock.h>

int plock (op)
int op;

DESCRIPTION

Plock allows the calling process to lock its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock) into memory. Locked segments are immune to all routine swapping. *Plock* also allows these segments to be unlocked. The effective user ID of the calling process must be super-user to use this call. *Op* specifies the following:

PROCLOCK –	lock text & data segments into memory (process lock)
TXTLOCK –	lock text segment into memory (text lock)
DATLOCK –	lock data segment into memory (data lock)
UNLOCK –	remove locks

Plock will fail and not perform the requested operation if one or more of the following are true:

The effective user ID of the calling process is not super-user. [EPERM]

Op is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process. [EINVAL]

Op is equal to **TXTLOCK** and a text lock, or a process lock already exists on the calling process. [EINVAL]

Op is equal to **DATLOCK** and a data lock, or a process lock already exists on the calling process. [EINVAL]

Op is equal to **UNLOCK** and no type of lock exists on the calling process. [EINVAL]

RETURN VALUE

Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2).

NAME

profil – execution time profile

SYNOPSIS

void profil (buff, bufsiz, offset, scale)
char *buff;
long bufsiz, offset;
unsigned scale;

DESCRIPTION

Buff points to an area of memory whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick (50th or 60th second); *offset* is subtracted from it, and the result shifted right by *scale*. If the resulting number corresponds to a word (2 bytes) inside *buff*, that word is incremented.

Profiling is turned off by giving a *scale* of 0. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *exec* is executed, but remains on in child and parent both after a *fork*. **Profil** will return -1, errno EINVAL if an update in *buff* would cause a memory fault.

RETURN VALUE

0 if ok, -1 if error.

SEE ALSO

prof(1), monitor(3C).

NAME

ptrace – process trace

SYNOPSIS

int ptrace (request, pid, addr, data);
int request, pid;
int * addr;
long data;

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process. Its primary use is for the implementation of breakpoint debugging; see *adb*(1). The child process behaves normally until it encounters a signal (see *signal*(2) for the list), at which time it enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its parent can examine and modify its "core image" using *ptrace*. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The *request* argument determines the precise action to be taken by *ptrace* and is one of the following:

0 This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by *func*; see *signal*(2). The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results will ensue if the parent does not expect to trace the child.

The remainder of the requests can only be used by the parent process. For each, *pid* is the process ID of the child. The child must be in a stopped state before these requests are made.

1, 2 With these requests, the word at location *addr* in the address space of the child is returned to the parent process. Request 1 returns a word from I space, and request 2 returns a word from D space. The *data* argument is ignored. These two requests will fail if *addr* is not the start address of a word, in which case a value of −1 is returned to the parent process and the parent's *errno* is set to ENXIO.

3 With this request, the word at location *addr* in the child's USER area in the system's address space (see **<sys/user.h>**) is returned to the parent process. Addresses in this area range from 0 to 3072 on the Cadmus 9000 and 0 to 2048 on the 3B20S and VAX. The *data* argument is ignored. This request will fail if *addr* is not the start address of a word or is outside the USER area, in which case a value of −1 is returned to the parent process and the parent's *errno* is set to EIO.

4, 5 With these requests, the low word of the value given by the *data* argument is written into the address space of the child at location *addr*. Request 4 writes a word into I space, and request 5 writes a word into D space. Upon successful completion, the value written into the address space of the child is returned to the parent. These two requests will fail if *addr* is a location in a pure procedure space and another process is executing in that space, or *addr* is not the start address of a word. Upon failure a value of −1 is returned to the parent process and the parent's *errno* is set to EIO.

6 With this request, a few entries in the child's USER area can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. The few entries that can be written are:

the general registers A7/D0-A6 on the Motorola 68000 (addr = 0..15),

the program counter PC (addr = 20),

and the low byte of the Processor Status Word PS (addr = 19).

For addrs 0..15 and 20, the whole long value of *data* is written, for addr = 19 only the least significant byte.

7 This request causes the child to resume execution. If the *data* argument is 0, all pending signals including the one that caused the child to stop are canceled before it resumes execution. If the *data* argument is a valid signal number, the child resumes execution as if it had incurred that signal and any other pending signals are canceled. If the *addr* argument is odd for this request, the program continues where it halted before. Otherwise, *addr* is made the new program counter. Upon successful completion, the value of *data* is returned to the parent.

8 This request causes the child to terminate with the same consequences as *exit*(2).

9 This request sets the trace bit in the Processor Status Word of the child (i.e., bit 0x8000 on the Motorola 68000) and then executes the same steps as listed above for request 7. The trace bit causes an interrupt upon completion of one machine instruction. This effectively allows single stepping of the child.

10 This request writes the user registers to *addr* as a record of the form struct exvec with variant ex2o. VECSIZE bytes will be transferred. See */usr/include/sys/reg.h*.

To forestall possible fraud, *ptrace* inhibits the set-user-id facility on subsequent *exec*(2) calls. If a traced process calls *exec*, it will stop before executing the first instruction of the new image showing signal SIGTRAP.

GENERAL ERRORS

Ptrace will in general fail if one or more of the following are true:

Request is an illegal number. [EIO]

Pid identifies a child that does not exist or has not executed a *ptrace* with request 0. [ESRCH]

SEE ALSO
adb(1), exec(2), signal(2), wait(2).

NAME

read – read from file

SYNOPSIS

int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line (see *ioctl*(2) and *termio*(4)), or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

When attempting to read from an empty pipe (or FIFO):

> If O_NDELAY is set, the read will return a 0.
>
> If O_NDELAY is clear, the read will block until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

> If O_NDELAY is set, the read will return a 0.
>
> If O_NDELAY is clear, the read will block until data becomes available.

Read will fail if one or more of the following are true:

> *Fildes* is not a valid file descriptor open for reading. [EBADF]
>
> *Buf* points outside the allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a –1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2), open(2), pipe(2), termio(4).

NAME

setpgrp – set process group ID

SYNOPSIS

int setpgrp ()

DESCRIPTION

Setpgrp sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

RETURN VALUE

Setpgrp returns the value of the new process group ID.

SEE ALSO

exec(2), fork(2), getpid(2), intro(2), kill(2), signal(2).

NAME

setuid, setgid — set user and group IDs

SYNOPSIS

int setuid (uid)
int uid;

int setgid (gid)
int gid;

DESCRIPTION

Setuid (*setgid*) is used to set the real user (group) ID and effective user (group) ID of the calling process.

If the effective user ID of the calling process is super-user, the real user (group) ID and effective user (group) ID are set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but its real user (group) ID is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

Setuid (*setgid*) will fail if the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user. [EPERM]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

getuid(2), intro(2).

NAME

signal – specify what to do upon receipt of a signal

SYNOPSIS

```
#include <sys/signal.h>

int (*signal (sig, func))()
int sig;
int (*func)();
```

DESCRIPTION

Signal allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *Sig* specifies the signal and *func* specifies the choice.

Sig can be assigned any one of the following except **SIGKILL**:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03*	quit
SIGILL	04*	illegal instruction (not reset when caught)
SIGTRAP	05*	trace trap (not reset when caught)
SIGIOT	06*	IOT instruction
SIGEMT	07*	EMT instruction
SIGFPE	08*	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGZERO	17	zero divide
SIGCHK	18	check error
SIGOVER	19	arithmetic overflow
SIGPRIV	20	privilege violation
SIGUSR1	21	user defined signal 1
SIGUSR2	22	user defined signal 2
SIGCLD	23	death of a child (see *WARNING* below)
SIGPWR	24	power fail (see *WARNING* below)

See below for the significance of the asterisk (*) in the above list.

Func is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values of are as follows:

SIG_DFL – terminate process upon receipt of a signal

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2) plus a "core image" will be made in the current working directory of the receiving process if *sig* is one for which an asterisk appears in the above list *and* the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask (see *umask*(2))

a file owner ID that is the same as the effective user ID of the receiving process

a file group ID that is the same as the effective group ID of the receiving process

SIG_IGN – ignore signal
The signal *sig* is to be ignored.

Note: the signal SIGKILL cannot be ignored.

function address – catch signal
Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function. Before entering the signal-catching function, the value of *func* for the caught signal will be set to SIG_DFL unless the signal is SIGILL, SIGTRAP, or SIGPWR.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted.

When a signal that is to be caught occurs during a *read*, a *write*, an *open*, or an *ioctl* system call on a slow device (like a terminal; but not a file), during a *pause* system call, or during a *wait* system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call will return a −1 to the calling process with *errno* set to EINTR.

Note: the signal SIGKILL cannot be caught.

A call to *signal* cancels a pending signal *sig* except for a pending SIGKILL signal.

Signal will fail if one or more of the following are true:

Sig is an illegal signal number, including SIGKILL. [EINVAL]

Func points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

kill(1), kill(2), pause(2), ptrace(2), wait(2), setjmp(3C).

WARNING

Two other signals that behave differently than the signals described above exist in this release of the system; they are:

SIGCLD	23	death of a child (reset when caught)
SIGPWR	24	power fail (not reset when caught)

There is no guarantee that, in future releases of the UNIX System, these signals will continue to behave as described below; they are included only for compatibility with other versions of the UNIX System. Their use in new programs is strongly discouraged. Note: signal SIGPWR is not yet supported on the Cadmus 9000.

For these signals, *func* is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address* . The actions prescribed by these values of are as follows:

SIG_DFL - ignore signal
The signal is to be ignored.

SIG_IGN - ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate; see *exit*(2).

function address - catch signal
If the signal is **SIGPWR**, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is **SIGCLD** except, that while the process is executing the signal-catching function any received **SIGCLD** signals will be queued and the signal-catching function will be continually reentered until the queue is empty.

The **SIGCLD** affects two other system calls (*wait*(2), and *exit*(2)) in the following ways:

wait If the *func* value of **SIGCLD** is set to **SIG_IGN** and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of −1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the *func* value of **SIGCLD** is set to **SIG_IGN**, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

NAME

stat, fstat – get file status

SYNOPSIS

```
#include <sys/types.h>

#include <sys/stat.h>

int stat (path, buf)

char *path;

struct stat *buf;

int fstat (fildes, buf)

int fildes;

struct stat *buf;
```

DESCRIPTION

Path points to a path name naming a file. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. *Stat* obtains information about the named file.

Similarly, *fstat* obtains information about an open file known by the file descriptor *fildes*, obtained from a successful *open*, *creat*, *dup*, *fcntl*, or *pipe* system call.

Buf is a pointer to a *stat* structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```
ushort  st_mode;        /* File mode; see mknod (2) */
ino_t   st_ino;         /* Inode number */
dev_t   st_dev;         /* ID of device containing */
                        /* a directory entry for this file */
dev_t   st_rdev;        /* ID of device */
                        /* This entry is defined only for */
                        /* character special or block special files */
short   st_nlink;       /* Number of links */
ushort  st_uid;         /* User ID of the file's owner */
ushort  st_gid;         /* Group ID of the file's group */
off_t   st_size;        /* File size in bytes */
time_t  st_atime;       /* Time of last access */
time_t  st_mtime;       /* Time of last data modification */
time_t  st_ctime;       /* Time of last file status change */
                        /* Times measured in seconds since */
                        /* 00:00:00 GMT, Jan. 1, 1970 */
```

`st_atime`	Time when file data was last accessed. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *read*(2).
`st_mtime`	Time when data was last modified. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *write*(2).
`st_ctime`	Time when file status was last changed. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *link*(2), *mknod*(2), *pipe*(2), *unlink*(2), *utime*(2), and *write*(2).

Stat will fail if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied for a component of the path prefix. [EACCES]

Buf or *path* points to an invalid address. [EFAULT]

Fstat will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Buf points to an invalid address. [EFAULT]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), chown(2), creat(2), link(2), mknod(2), time(2), unlink(2).

NAME

stime – set time

SYNOPSIS

```
int stime (tp)
long *tp;
```

DESCRIPTION

Stime sets the system's idea of the time and date. *Tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

Stime will fail if the effective user ID of the calling process is not super-user. [EPERM]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

time(2).

NAME

sync – update super-block

SYNOPSIS

void sync ( )

DESCRIPTION

Sync causes all information in memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *fsck*, *df*, etc. It is mandatory before a boot.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

NAME

time − get time

SYNOPSIS

long time ((long *) 0)

long time (tloc)
long *tloc;

DESCRIPTION

Time returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

If *tloc* (taken as an integer) is non-zero, the return value is also stored in the location to which *tloc* points.

Time will fail if *tloc* points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, *time* returns the value of time. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

stime(2).

NAME

times – get process and child process times

SYNOPSIS

```
#include <sys/types.h>

#include <sys/times.h>

long times (buffer)

struct tms *buffer;
```

DESCRIPTION

Times fills the structure pointed to by ***buffer*** with time-accounting information. The following is this contents of the structure:

```
struct  tms {
        time_t  tms_utime;
        time_t  tms_stime;
        time_t  tms_cutime;
        time_t  tms_cstime;
};
```

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*. All times are in 60ths of a second on DEC processors, 100ths of a second on WECo processors.

Tms_utime is the CPU time used while executing instructions in the user space of the calling process.

Tms_stime is the CPU time used by the system on behalf of the calling process.

Tms_cutime is the sum of the *tms_utime*s and *tms_cutime*s of the child processes.

Tms_cstime is the sum of the *tms_stime*s and *tms_cstime*s of the child processes.

Times will fail if ***buffer*** points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, *times* returns the elapsed real time, in 60ths (100ths) of a second, since an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, a –1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), fork(2), time(2), wait(2).

NAME

ulimit – get and set user limits

SYNOPSIS

long ulimit (cmd, newlimit)
int cmd;
long newlimit;

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

1 Get the process's file size limit. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

2 Set the process's file size limit to the value of ***newlimit***. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. *Ulimit* will fail and the limit will be unchanged if a process with an effective user ID other than super-user attempts to increase its file size limit. [EPERM]

3 Get the maximum possible break value. See *brk*(2).

RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

brk(2), write(2).

NAME

umask – set and get file creation mask

SYNOPSIS

int umask (cmask)
int cmask;

DESCRIPTION

Umask sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

RETURN VALUE

The previous value of the file mode creation mask is returned.

SEE ALSO

mkdir(1), sh(1), chmod(2), creat(2), mknod(2), open(2).

NAME

umount – unmount a file system

SYNOPSIS

int umount (spec)
char *spec;

DESCRIPTION

Umount requests that a previously mounted file system contained on the block special device identified by *spec* be unmounted. *Spec* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

Umount may be invoked only by the super-user.

Umount will fail if one or more of the following are true:

The process's effective user ID is not super-user. [EPERM]

Spec does not exist. [ENXIO]

Spec is not a block special device. [ENOTBLK]

Spec is not mounted. [EINVAL]

A file on *spec* is busy. [EBUSY]

Spec points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

mount(2).

NAME

uname, ethname – get name/ethernet-identification of current UNIX system

SYNOPSIS

```
#include <sys/utsname.h>

int uname (name)

struct utsname *name;

int ethname (name)

struct ethname *name;
```

DESCRIPTION

Uname stores information identifying the current UNIX system in the structure pointed to by *name*.

Uname uses the structure defined in <sys/utsname.h> whose members are:

```
char    sysname[9];
char    nodename[9];
char    release[9];
char    version[9];
```

Uname returns a null-terminated character string naming the current UNIX system in the character array *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. *Release* and *version* further identify the operating system.

Ethname stores information identifying the ethernet address and station number in the structure pointed to by *name*.

Ethname uses the structure ethname defined in <sys/utsname.h> whose members are:

```
etheradr        ethaddr;
short           stnaddr;
```

ethaddr contains the six byte long ethernet address. *stnaddr* is the station number, a small unique integer for each system on the net.

Uname and ethname will fail if *name* points to an invalid address. [EFAULT]

RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, –1 is returned and *errno* is set to indicate the error.

SEE ALSO

uname(1).

NAME

unlink – remove directory entry

SYNOPSIS

int unlink (path)
char *path;

DESCRIPTION

Unlink removes the directory entry named by the path name pointed to be *path*.

The named file is unlinked unless one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied for a component of the path prefix. [EACCES]

Write permission is denied on the directory containing the link to be removed. [EACCES]

The named file is a directory and the effective user ID of the process is not super-user. [EPERM]

The entry to be unlinked is the mount point for a mounted file system. [EBUSY]

The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed. [ETXTBSY]

The directory entry to be unlinked is part of a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

rm(1), close(2), link(2), open(2).

NAME

ustat – get file system statistics

SYNOPSIS

```
#include <sys/types.h>

#include <ustat.h>

int ustat (dev, buf)

int dev;

struct ustat *buf;
```

DESCRIPTION

Ustat returns information about a mounted file system. *Dev* is a device number identifying a device containing a mounted file system. *Buf* is a pointer to a *ustat* structure that includes the following elements:

```
daddr_t f_tfree;        /* Total free blocks */
ino_t   f_tinode;       /* Number of free inodes */
char    f_fname[6];     /* Filsys name */
char    f_fpack[6];     /* Filsys pack name */
```

Ustat will fail if one or more of the following are true:

Dev is not the device number of a device containing a mounted file system. [EINVAL]

Buf points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

stat(2), fs(4).

NAME

utime – set file access and modification times

SYNOPSIS

```
#include <sys/types.h>

int utime (path, times)

char *path;

struct utimbuf *times;
```

DESCRIPTION

Path points to a path name naming a file. *Utime* sets the access and modification times of the named file.

If *times* is NULL, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use *utime* in this manner.

If *times* is not NULL, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user may use *utime* this way.

The times in the following structure are measured in seconds since 00:00:00 GMT, Jan. 1, 1970.

```
struct  utimbuf {
        time_t  actime;         /* access time */
        time_t  modtime;        /* modification time */
};
```

Utime will fail if one or more of the following are true:

The named file does not exist. [ENOENT]

A component of the path prefix is not a directory. [ENOTDIR]

Search permission is denied by a component of the path prefix. [EACCES]

The effective user ID is not super-user and not the owner of the file and *times* is not NULL. [EPERM]

The effective user ID is not super-user and not the owner of the file and *times* is NULL and write access is denied. [EACCES]

The file system containing the file is mounted read-only. [EROFS]

Times is not NULL and points outside the process's allocated address space. [EFAULT]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

stat(2).

NAME

wait – wait for child process to stop or terminate

SYNOPSIS

int wait (stat_loc)
int *stat_loc;

int wait ((int *)0)

DESCRIPTION

Wait suspends the calling process until it receives a signal that is to be caught (see *signal*(2)), or until any one of the calling process's child processes stops in a trace mode (see *ptrace*(2)) or terminates. If a child process stopped or terminated prior to the call on *wait*, return is immediate.

If *stat_loc* (taken as an integer) is non-zero, 16 bits of information called status are stored in the low order 16 bits of the location pointed to by *stat_loc*. *Status* can be used to differentiate between stopped and terminated child processes and if the child process terminated, status identifies the cause of termination and pass useful information to the parent. This is accomplished in the following manner:

> If the child process stopped, the high order 8 bits of status will contain the number of the signal that caused the process to stop and the low order 8 bits will be set equal to 0177.
>
> If the child process terminated due to an *exit* call, the low order 8 bits of status will be zero and the high order 8 bits will contain the low order 8 bits of the argument that the child process passed to *exit*; see *exit*(2).
>
> If the child process terminated due to a signal, the high order 8 bits of status will be zero and the low order 8 bits will contain the number of the signal that caused the termination. In addition, if the low order seventh bit (i.e., bit 200) is set, a "core image" will have been produced; see *signal*(2).

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes; see *intro*(2).

Wait will fail and return immediately if one or more of the following are true:

> The calling process has no existing unwaited-for child processes. [ECHILD]
>
> *Stat_loc* points to an illegal address. [EFAULT]

RETURN VALUE

If *wait* returns due to the receipt of a signal, a value of −1 is returned to the calling process and *errno* is set to EINTR. If *wait* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2), pause(2), signal(2).

WARNING

See *WARNING* in *signal*(2).

NAME

write – write on a file

SYNOPSIS

```
int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

Write will fail and the file pointer will remain unchanged if one or more of the following are true:

> *Fildes* is not a valid file descriptor open for writing. [EBADF]
>
> An attempt is made to write to a pipe that is not open for reading by any process. [EPIPE and SIGPIPE signal]
>
> An attempt was made to write a file that exceeds the process's file size limit or the maximum file size. See *ulimit*(2). [EFBIG]
>
> *Buf* points outside the process's allocated address space. [EFAULT]

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* (see *ulimit*(2)) or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

If the file being written is a pipe (or FIFO), no partial writes will be permitted. Thus, the write will fail if a write of *nbyte* bytes would exceed a limit.

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then write to a full pipe (or FIFO) will return a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe (or FIFO) will block until space becomes available.

RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise, −1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), lseek(2), open(2), pipe(2), ulimit(2).

NAME

intro – introduction to subroutines and libraries

SYNOPSIS

#include <stdio.h>

#include <math.h>

DESCRIPTION

This section describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2 of this volume. Certain major collections are identified by a letter after the section number:

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library *libc*, which is automatically loaded by the C compiler, *cc*(1). The link editor *ld*(1) searches this library under the **–lc** option. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.

(3F) These functions constitute the FORTRAN intrinsic function library, *libF77*. These functions are automatically available to the FORTRAN programmer and require no special invocation of the compiler.

(3M) These functions constitute the Math Libraries, *libffp*,*libmot*,*libnsc*. They are automatically loaded as needed by the FORTRAN compiler *f77*(1). They are automatically loaded by the C compiler, *cc*(1); if the options **–f**,**–fF**, or**–fN** are given. Declarations for these functions may be obtained from the **#include** file **<math.h>**.

(3S) These functions constitute the "standard I/O package" (see *stdio*(3S)). These functions are in the library *libc*, already mentioned. Declarations for these functions may be obtained from the **#include** file **<stdio.h>**.

(3X) Various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

DEFINITIONS

A *character* is any bit pattern able to fit into a byte on the machine. The *null character* is a character with value 0, represented in the C language as '\0'. 'A *character array* is a sequence of characters. A *null-terminated character array* is a sequence of characters, the last of which is the *null character*. A *string* is a designation for a *null-terminated character array*. The *null string* is a character array containing only the null character. A **NULL** pointer is the value that is obtained by casting **0** into a pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error. **NULL** is defined as **(char *)0** in **<stdio.h>**; the user can include his own definition if he is not using **<stdio.h>**.

Many groups of FORTRAN intrinsic functions have *generic* function names that do not require explicit or implicit type declaration. The type of the function will be determined by the type of its argument(s). For example, the generic function *max* will return an integer value if given integer arguments (*max0*), a real value if given real arguments (*amax1*), or a double-precision value if given double-precision arguments (*dmax1*).

FILES

/lib/libc.a
/usr/lib/libF77.a
/lib/libffp.a
/lib/libmot.a
/lib/libnsc.a

SEE ALSO

ar(1), cc(1), f77(1), ld(1), nm(1), intro(2), stdio(3S).

DIAGNOSTICS

Functions in the Math Library (3M) may return the conventional values 0 or HUGE (the largest single-precision floating-point number) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable *errno* (see *intro*(2)) is set to the value EDOM or ERANGE. As many of the FORTRAN intrinsic functions use the routines found in the Math Library, the same conventions apply.

NAME

a64l, l64a – convert between long and base-64 ASCII

SYNOPSIS

```
long a64l (s)
char *s;

char *l64a (l)
long l;
```

DESCRIPTION

These routines are used to maintain numbers stored in *base-64* ASCII. This is a notation by which long integers can be represented by up to six characters; each character represents a "digit" in a radix-64 notation.

The characters used to represent "digits" are **.** for 0, **/** for 1, **0** through **9** for 2–11, **A** through **Z** for 12–37, and **a** through **z** for 38–63.

A64l takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. *L64a* takes a **long** argument and returns a pointer to the corresponding base-64 representation.

BUGS

The value returned by *l64a* is a pointer into a static buffer, the contents of which are overwritten by each call.

NAME

abort – generate an IOT fault

SYNOPSIS

abort ( )

DESCRIPTION

Abort causes an IOT signal to be sent to the process. This usually results in termination with a core dump.

It is possible for *abort* to return control if SIGIOT is caught or ignored.

SEE ALSO

adb(1), exit(2), signal(2).

DIAGNOSTICS

Usually "abort – core dumped" from the shell.

NAME

abort – terminate Fortran program

SYNOPSIS

call abort ( )

DESCRIPTION

Abort terminates the program which calls it, closing all open files truncated to the current position of the file pointer.

DIAGNOSTICS

When invoked, *abort* prints "Fortran abort routine called" on the standard error output.

SEE ALSO

abort(3C).

NAME

abs – integer absolute value

SYNOPSIS

int abs (i)
int i;

DESCRIPTION

Abs returns the absolute value of its integer operand.

SEE ALSO

fabs(3M).

BUGS

You get what the hardware gives on the largest negative integer.

NAME

abs, iabs, dabs, cabs, zabs – Fortran absolute value

SYNOPSIS

```
integer i1, i2
real r1, r2
double precision dp1, dp2
complex cx1, cx2
double complex dx1, dx2

r2 = abs(r1)

i2 = iabs(i1)
i2 = abs(i1)

dp2 = dabs(dp1)
dp2 = abs(dp1)

cx2 = cabs(cx1)
cx2 = abs(cx1)

dx2 = zabs(dx1)
dx2 = abs(dx1)
```

DESCRIPTION

Abs is the family of absolute value functions. *Iabs* returns the integer absolute value of its integer argument. *Dabs* returns the double-precision absolute value of its double-precision argument. *Cabs* returns the complex absolute value of its complex argument. *Zabs* returns the double-complex absolute value of its double-complex argument. The generic form *abs* returns the type of its argument.

SEE ALSO

floor(3M).

NAME

acos, dacos – Fortran arccosine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = acos(r1)

dp2 = dacos(dp1)
dp2 = acos(dp1)

DESCRIPTION

Acos returns the real arccosine of its real argument. *Dacos* returns the double-precision arccosine of its double-precision argument. The generic form *acos* may be used with impunity as its argument will determine the type of the returned value.

SEE ALSO

trig(3M).

NAME

aimag, dimag – Fortran imaginary part of complex argument

SYNOPSIS

```
real r
complex cxr
double precision dp
double complex cxd

r = aimag(cxr)

dp = dimag(cxd)
```

DESCRIPTION

Aimag returns the imaginary part of its single-precision complex argument. *Dimag* returns the double-precision imaginary part of its double-complex argument.

NAME

aint, dint – Fortran integer part intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = aint(r1)

dp2 = dint(dp1)
dp2 = aint(dp1)

DESCRIPTION

Aint returns the truncated value of its real argument in a real. *Dint* returns the truncated value of its double-precision argument as a double-precision value. *Aint* may be used as a generic function name, returning either a real or double-precision value depending on the type of its argument.

NAME

asin, dasin – Fortran arcsine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = asin(r1)

dp2 = dasin(dp1)
dp2 = asin(dp1)

DESCRIPTION

Asin returns the real arcsine of its real argument. *Dasin* returns the double-precision arcsine of its double-precision argument. The generic form *asin* may be used with impunity as it derives its type from that of its argument.

SEE ALSO

trig(3M).

NAME

assert – verify program assertion

SYNOPSIS

#include <assert.h>

assert (expression)
int expression;

DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), *assert* prints

"Assertion failed: *expression*, file *xyz*, line *nnn*"

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the *assert* statement.

Compiling with the preprocessor option **–DNDEBUG** (see *cpp*(1)), or with the preprocessor control statement "#define NDEBUG" ahead of the "#include <assert.h>" statement, will stop assertions from being compiled into the program.

SEE ALSO

cpp(1), abort(3C).

NAME

atan, datan – Fortran arctangent intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2

r2 = atan(r1)

dp2 = datan(dp1)
dp2 = atan(dp1)
```

DESCRIPTION

Atan returns the real arctangent of its real argument. *Datan* returns the double-precision arctangent of its double-precision argument. The generic form *atan* may be used with a double-precision argument returning a double-precision value.

SEE ALSO

trig(3M).

NAME

atan2, datan2 – Fortran arctangent intrinsic function

SYNOPSIS

```
real r1, r2, r3
double precision dp1, dp2, dp3

r3 = atan2(r1, r2)

dp3 = datan2(dp1, dp2)
dp3 = atan2(dp1, dp2)
```

DESCRIPTION

Atan2 returns the arctangent of *arg1/arg2* as a real value. *Datan2* returns the double-precision arctangent of its double-precision arguments. The generic form *atan2* may be used with impunity with double-precision arguments.

SEE ALSO

trig(3M).

NAME

atof, atoi, atol — convert ASCII to numbers

SYNOPSIS

double atof (nptr)
char *nptr;

int atoi (nptr)
char *nptr;

long atol (nptr)
char *nptr;

DESCRIPTION

These functions convert a string pointed to by *nptr* to floating, integer, and long integer representation respectively. The first unrecognized character ends the string.

Atof recognizes an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optionally signed integer.

Atoi and *atol* recognize an optional string of tabs and spaces, then an optional sign, then a string of digits.

SEE ALSO

scanf(3S).

BUGS

There are no provisions for overflow.

NAME

j0, j1, jn, y0, y1, yn – Bessel functions

SYNOPSIS

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
int n;
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n;
double x;
```

DESCRIPTION

J0 and *j1* return Bessel functions of *x* of the first kind of orders 0 and 1 respectively. *Jn* returns the Bessel function of *x* of the first kind of order *n*.

Y0 and *y1* return the Bessel functions of *x* of the second kind of orders 0 and 1 respectively. *Yn* returns the Bessel function of *x* of the second kind of order *n*. The value of *x* must be positive.

DIAGNOSTICS

Non-positive arguments cause *y0*, *y1* and *yn* to return the value **HUGE**, and to set *errno* to **EDOM**. They also cause a message indicating DOMAIN error to be printed on the standard error output; the process will continue.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

Bip – basic functions for BIP, the PCS bitmap display

SYNOPSIS

Bip can be used either as a normal terminal, /dev/bip, or as a graphics terminal, with the local intelligence described below.

As a normal dumb terminal, Bip emulates a vt100 with 66 lines, 113 columns, and the control sequences given in <bip/vt100.h>, /etc/termcap and /etc/keycap. The Unix 1.5 versions of med and vi can run on this /dev/bip. To make bip a normal login tty, see man 5 inittab.

As a graphics terminal, Bip uses programs in its local M68000 processor (either 'soft prom' or 'hard prom') to do the following graphics functions:

```
    Bdot( x, y, paint )  – paint is one of the 16 "painting rules" below.
    Bline( x0, y0, x1, y1, paint )
    Bcircle( x0, y0, radius, paint, 0 )
    Blt( x, y, h, w,  xto, yto,  paint )  – block transfer:
         x, y, h, w  are the block's upper left corner, height, and width
    Bfill( x, y, h, w, paint )
    Bget(  x, y, h, w, mem )   int mem[];
    Bput(  x, y, h, w, mem, paint )
    Bwrite( buf, len )  – vt100 emulation, as for >/dev/bip
Fputc( ch, x, y, font )  ^put a char from Berkeley 'font  (not yet)
Bipinit( "/dev/bip" )  – call before anything else.
```

DESCRIPTION

Blt, Bget, Bput ... move blocks on the Bip screen, or between Bip and main memory. For example, high-quality text is displayed by Bput ting (say) 12 by 20 bits for each letter, from memory to the screen.

The x coordinate runs 0..1023 left to right, and y runs 0..799 top to bottom. The top left corner is 0, 0, the bottom left is 0, 799. y actually runs 0..1022, but only the top 800 are displayed; the 223 off-screen rows can be used for storing e.g. fonts. The last y row 1023 is Bip control registers; don't write it.

The mem argument for Bput and Bget is a pointer to an array of bits: top row left-to-right, 2nd row left-to-right, ... mem is w/16 short words wide by h bits high. That is, if w <= 16, mem is an array of short s, if 16 < w <= 32 an array of long s, and so on. The leftmost (most significant) bit in mem[0] is painted at x, y, the next at x+1, y and so on.

Bip can "paint" source over destination in various ways, such as "black" or "white" or "invert" or "source over destination". There are 16 possible painting rules or "raster ops", encoded in 4 bits. For example, to 'or' source to destination:

```
    Bsource:          0  0  1  1
    Bdest:            0  1  0  1
    ____________________________
    Bsource | Bdest:  0  1  1  1
```

is paint 0111 == 7. 0 is black and 1 white, so this paint 7 would be used to 'or' white letters onto a black screen; to 'or' black-on-white letters onto a white background, Bput(... ~ (Bsource | ~Bdest)) .

FILES

`#include <bip/ops.h>`
: #defines the painting rules or "raster ops" Bblack, Bwhite, Bdest and Bsource, as well as Bhi 800 and Bwide 1024.

`cc ... -lbip`
: uses the Bip function library (full name /usr/lib/libbip.a). (The real Bit etc. functions are in prom; the library Bit just moves its arguments to local 68000 ram, then calls the prom Bit).

`cc ... -lqbip`
: has Qbus versions of the functions, which go directly host → pixel-processor, without the local 68000.

Bip system:

```
      QU68000 host
          |
          | Qbus: host memory <-> Bip memory
          |

      Bip: an M68000 with 128k ram, 16k rom, and pixel processor
          |
          |

      1024 * 1023 'bit map' or 2-port memory
          |
          |
      ...............................
      ...............................
      ....... screen .........
      ...............................
      ...............................
```

EXAMPLES

```
B⇓l( 0, 0, Bhi, Bwide,  Bwhite );  – clear the screen
Bline( 0, Bhi-1, Bwide-1, 0, Binvert );  – a diagonal line
...
```

REFERENCES

For an introduction to Bit and its use in graphics, see the excellent article by D. Ingalls, "The Smalltalk Graphics Kernel", in August 1981 Byte.

NAME

and, or, xor, not, lshift, rshift – Fortran bitwise boolean functions

SYNOPSIS

```
integer i, j, k
real a, b, c
double precision dp1, dp2, dp3

k = and(i, j)
c = or(a, b)
j = xor(i, a)
j = not(i)
k = lshift(i, j)
k = rshift(i, j)
```

DESCRIPTION

The generic intrinsic boolean functions *and*, *or* and *xor* return the value of the binary operations on their arguments. *Not* is a unary operator returning the one's complement of its argument. *Lshift* and *rshift* return the value of the first argument shifted left or right, respectively, the number of times specified by the second (integer) argument.

The boolean functions are generic, that is, they are defined for all data types as arguments and return values. Where required, the compiler will generate appropriate type conversions.

NOTE

Although defined for all data types, use of boolean functions on any but integer data is bizarre and will probably result in unexpected consequences.

BUGS

The implementation of the shift functions may cause large shift values to deliver weird results.

NAME

bsearch — binary search

SYNOPSIS

char *bsearch ((char *) key, (char *) base, nel, sizeof (*key), compar)
unsigned nel;
int (*compar)( );

DESCRIPTION

Bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *Key* points to the datum to be sought in the table. *Base* points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according as the first argument is to be considered less than, equal to, or greater than the second.

DIAGNOSTICS

A **NULL** pointer is returned if the key cannot be found in the table.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

lsearch(3C), hsearch(3C), qsort(3C), tsearch(3C).

NAME

clock – report CPU time used

SYNOPSIS

long clock ( )

DESCRIPTION

Clock returns the amount of CPU time (in microseconds) used since the first call to *clock*. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed *wait*(2) or *system*(3S).

The resolution of the clock is 1000/HZ milliseconds, where HZ = 50 or 60 is the line frequency.

SEE ALSO

times(2), wait(2), hertz(2), system(3S).

BUGS

The value returned by *clock* is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned will wrap around after accumulating only 2147 seconds of CPU time (about 36 minutes).

NAME

conjg, dconjg – Fortran complex conjugate intrinsic function

SYNOPSIS

complex cx1, cx2
double complex dx1, dx2

cx2 = conjg(cx1)

dx2 = dconjg(dx1)

DESCRIPTION

Conjg returns the complex conjugate of its complex argument. *Dconjg* returns the double-complex conjugate of its double-complex argument.

NAME

toupper, tolower, toascii – character translation

SYNOPSIS

```
#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;
```

DESCRIPTION

Toupper and *tolower* have as domain the range of *getc*: the integers from −1 through 255. If the argument of *toupper* represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

_toupper and *_tolower* are macros that accomplish the same thing as *toupper* and *tolower* but have restricted domains and are faster. *_toupper* requires a lower-case letter as its argument; its result is the corresponding upper-case letter. *_tolower* requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause garbage results.

Toascii yields its argument with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

SEE ALSO

ctype(3C).

NAME

cos, dcos, ccos – Fortran cosine intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2 = cos(r1)

dp2 = dcos(dp1)
dp2 = cos(dp1)

cx2 = ccos(cx1)
cx2 = cos(cx1)
```

DESCRIPTION

Cos returns the real cosine of its real argument. *Dcos* returns the double-precision cosine of its double-precision argument. *Ccos* returns the complex cosine of its complex argument. The generic form *cos* may be used with impunity as its returned type is determined by that of its argument.

SEE ALSO

trig(3M).

NAME

cosh, dcosh – Fortran hyperbolic cosine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = cosh(r1)

dp2 = dcosh(dp1)
dp2 = cosh(dp1)

DESCRIPTION

Cosh returns the real hyperbolic cosine of its real argument. *Dcosh* returns the double-precision hyperbolic cosine of its double-precision argument. The generic form *cosh* may be used to return the hyperbolic cosine in the type of its argument.

SEE ALSO

sinh(3M).

NAME

crypt, setkey, encrypt – DES encryption

SYNOPSIS

```
char *crypt (key, salt)
char *key, *salt;

setkey (key)
char *key;

encrypt (block, edflag)
char *block;
int edflag;
```

DESCRIPTION

Crypt is the password encryption routine. It is based on the NBS Data Encryption Standard (DES), with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

The first argument to *crypt* is a user's typed password. The second is a 2-character string chosen from the set **[a-zA-Z0-9./]**; this *salt* string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the salt. The first two characters are the salt itself.

The *setkey* and *encrypt* entries provide (rather primitive) access to the actual DES algorithm. The argument of *setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored, leading to a 56-bit key which is set into the machine.

The argument to the *encrypt* entry is likewise a character array of length 64 containing 0's and 1's. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by *setkey*. If *edflag* is 0, the argument is encrypted; if non-zero, it is decrypted.

SEE ALSO

login(1), passwd(1), getpass(3C), passwd(5).

BUGS

The return value points to static data that are overwritten by each call.

NAME

ctermid – generate file name for terminal

SYNOPSIS

#include <stdio.h>

char *ctermid(s)
char *s;

DESCRIPTION

Ctermid generates a string that refers to the controlling terminal for the current process when used as a file name.

If (int)*s* is zero, the string is stored in an internal static area, the contents of which are overwritten at the next call to *ctermid*, and the address of which is returned. If (int)*s* is non-zero, then *s* is assumed to point to a character array of at least **L_ctermid** elements; the string is placed in this array and the value of *s* is returned. The manifest constant **L_ctermid** is defined in **<stdio.h>**.

NOTES

The difference between *ctermid* and *ttyname*(3C) is that *ttyname* must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a magic string (**/dev/tty**) that will refer to the terminal if used as a file name. Thus *ttyname* is useless unless the process already has at least one file open to a terminal.

SEE ALSO

ttyname(3C).

NAME

ctime, localtime, gmtime, asctime, tzset – convert date and time to ASCII

SYNOPSIS

```
char *ctime (clock)

long *clock;

#include <time.h>

struct tm *localtime (clock)

long *clock;

struct tm *gmtime (clock)

long *clock;

char *asctime (tm)

struct tm *tm;

tzset ( )
```

DESCRIPTION

Ctime converts a time pointed to by *clock* such as returned by *time*(2) into ASCII and returns a pointer to a 26-character string in the following form. All the fields have constant width.

    Sun Sep 16 01:03:52 1973\n\0

Localtime and *gmtime* return pointers to structures containing the broken-down time. *Localtime* corrects for the time zone and possible daylight savings time; *gmtime* converts directly to GMT, which is the time the UNIX system uses. *Asctime* converts a broken-down time to ASCII and returns a pointer to a 26-character string.

The structure declaration from the include file is:

```
/*      @(#)time.h      1.1       */
/*      3.0 SID #       1.2       */
struct  tm {    /* see ctime(3) */
        int     tm_sec;
        int     tm_min;
        int     tm_hour;
        int     tm_mday;
        int     tm_mon;
        int     tm_year;
        int     tm_wday;
        int     tm_yday;
        int     tm_isdst;
};
extern struct tm *gmtime(), *localtime();
extern char *ctime(), *asctime();
extern void tzset();
extern long timezone;
extern int daylight;
extern char *tzname[];
```

These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday = 0), year – 1900, day of year (0-365), and a flag that is non-zero if daylight saving time is in effect.

The external long variable *timezone* contains the difference, in seconds, between GMT and local standard time (in EST, *timezone* is 5•60•60); the external variable *daylight* is non-zero if and only if the standard U.S.A.

Daylight Savings Time conversion should be applied. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

If an environment variable named TZ is present, *asctime* uses the contents of the variable to override the default time zone. The value of TZ must be a three-letter time zone name, followed by a number representing the difference between local time and Greenwich time in hours, followed by an optional three-letter name for a daylight time zone. For example, the setting for New Jersey would be EST5EDT. The effects of setting TZ are thus to change the values of the external variables *timezone* and *daylight*; in addition, the time zone names contained in the external variable

```
char *tzname[2] = {"EST", "EDT"};
```

are set from the environment variable. The function *tzset* sets the external variables from TZ; it is called by *asctime* and may also be called explicitly by the user.

SEE ALSO

time(2), getenv(3C), environ(7).

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii – classify characters

SYNOPSIS

#include <ctype.h>

int isalpha (c)
int c;

. . .

DESCRIPTION

These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *Isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value **EOF** (–1 – see *stdio*(3S)).

isalpha	c is a letter.
isupper	c is an upper-case letter.
islower	c is a lower-case letter.
isdigit	c is a digit [0-9].
isxdigit	c is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	c is an alphanumeric (letter or digit).
isspace	c is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	c is a punctuation character (neither control nor alphanumeric).
isprint	c is a printing character, code 040 (space) through 0176 (tilde).
isgraph	c is a printing character, like *isprint* except false for space.
iscntrl	c is a delete character (0177) or an ordinary control character (less than 040).
isascii	c is an ASCII character, code less than 0200.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

SEE ALSO

ascii(7).

NAME
curses – screen functions with "optimal" cursor motion

SYNOPSIS
cc [flags] files **–lcurses –ltermcap** [libraries]

DESCRIPTION
These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the *refresh()* tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine *initscr()* must be called before any of the other routines that deal with windows and screens are used. The routine *endwin()* should be called before exiting.

SEE ALSO
Screen Updating and Cursor Movement Optimization: A Library Package, Ken Arnold,
ioctl(2), getenv(3), termcap(5)

AUTHOR
Ken Arnold

FUNCTIONS

addch(ch)	add a character to *stdscr*
addstr(str)	add a string to *stdscr*
box(win,vert,hor)	draw a box around a window
crmode()	set cbreak mode
clear()	clear *stdscr*
clearok(scr,boolf)	set clear flag for *scr*
clrtobot()	clear to bottom on *stdscr*
clrtoeol()	clear to end of line on *stdscr*
delch()	delete a character
deleteln()	delete a line
delwin(win)	delete *win*
echo()	set echo mode
endwin()	end window modes
erase()	erase *stdscr*
getch()	get a char through *stdscr*
getcap(name)	get terminal capability *name*
getstr(str)	get a string through *stdscr*
gettmode()	get tty modes
getyx(win,y,x)	get (y,x) co-ordinates
inch()	get char at current (y,x) co-ordinates
initscr()	initialize screens
insch(c)	insert a char
insertln()	insert a line
leaveok(win,boolf)	set leave flag for *win*
longname(termbuf,name)	get long name from *termbuf*
move(y,x)	move to (y,x) on *stdscr*
mvcur(lasty,lastx,newy,newx)	actually move cursor
newwin(lines,cols,begin_y,begin_x)	create a new window
nl()	set newline mapping
nocrmode()	unset cbreak mode
noecho()	unset echo mode

nonl()	unset newline mapping
noraw()	unset raw mode
overlay(win1,win2)	overlay win1 on win2
overwrite(win1,win2)	overwrite win1 on top of win2
printw(fmt,arg1,arg2,...)	printf on *stdscr*
raw()	set raw mode
refresh()	make current screen look like *stdscr*
resetty()	reset tty flags to stored value
savetty()	stored current tty flags
scanw(fmt,arg1,arg2,...)	scanf through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win,boolf)	set scroll flag
setterm(name)	set term variables for name
standend()	end standout mode
standout()	start standout mode
subwin(win,lines,cols,begin_y,begin_x)	create a subwindow
touchwin(win)	change all of *win*
unctrl(ch)	printable version of *ch*
waddch(win,ch)	add char to *win*
waddstr(win,str)	add string to *win*
wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*
wdelch(win,c)	delete char from *win*
wdeleteln(win)	delete line from *win*
werase(win)	erase *win*
wgetch(win)	get a char through *win*
wgetstr(win,str)	get a string through *win*
winch(win)	get char at current (y,x) in *win*
winsch(win,c)	insert char into *win*
winsertln(win)	insert line into *win*
wmove(win,y,x)	set current (y,x) co-ordinates on *win*
wprintw(win,fmt,arg1,arg2,...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win,fmt,arg1,arg2,...)	scanf through *win*
wstandend(win)	end standout mode on *win*
wstandout(win)	start standout mode on *win*

NAME

cuserid – character login name of the user

SYNOPSIS

```
#include <stdio.h>

char *cuserid (s)
char *s;
```

DESCRIPTION

Cuserid generates a character representation of the login name of the owner of the current process. If (int)*s* is zero, this representation is generated in an internal static area, the address of which is returned. If (int)*s* is non-zero, *s* is assumed to point to an array of at least **L_cuserid** characters; the representation is left in this array. The manifest constant **L_cuserid** is defined in **<stdio.h>**.

DIAGNOSTICS

If the login name cannot be found, *cuserid* returns **NULL**; if *s* is non-zero in this case, **\0** will be placed at **s*.

SEE ALSO

getlogin(3C), getpwuid(3C).

BUGS

Cuserid uses *getpwnam*(3C); thus the results of a user's call to the latter will be obliterated by a subsequent call to the former.
The name *cuserid* is rather a misnomer.

NAME

d – text database functions

DESCRIPTION

d is a set of functions that work on Unix text files with records' like:

```
        from: Smith     to: Jones     date: 15.5     re: Unix jobs
```

or:

Title:	What is the Title of this Book?
Author:	R.M.Smullyan
Year:	1978
Subjects:	logic, games, paradox

or Usenet mail, or a software project database like [Knudsen].

d is designed for small text databases like these, where

- the familiar, fast screen editor can edit the data
- Unix tools `sort, grep, awk, join ...` all work
- the size and cost of a big database system would be overkill.

d line, d record, d get, d form:

d has four subcommands:

```
d  {line | record | get | form}   [options]   [files, or stdin]
```

`line, record, get, form` may be abbreviated `l, r, g, f` .

`d line`

makes each record into a single 'line', for `sort` or `awk` or ...

`d record`

undoes `d line` in:

```
        d line afile   |  Unix-filter   |  d record
```

```
where   d line          flattens each record to one line,
        Unix-filter     sorts or awks ... the lines, and
        d record        makes each line back into a record.
```

`d get` aword

gets all records contining 'aword', like a fast
`d line | fgrep aword | d record`

`d form` aformfile

puts fields from the data base into a form, such as:

    Dear {Name},
        In reply to your {Adjective} letter of {Date},
        ...

or simply selects some fields:

    {Phone} {Name} .

EXAMPLES

`d get Edinburgh eunet.map`

looks in the `eunet` map for people in Edinburgh.

```
d line < project | sort '-t|' -r +2 | d rec
```
sorts project records on field 3.

d record format

A record is a group of lines, followed by a blank line. A field is 'Fieldname: Value', where 'Value' is everything up to the next 'Fieldname': a word, a line, or several lines. 'Value' may contain a ':', but not right after a letter or number.

`d line` separates fields with a '|' field separator (`awk FS`, `sort -t`). `d record` then deletes all '|'s, so that `d line | d record` does nothing.

`d line` also changes newlines in multi-line records to an ersatz newline, default '\', which `d record` changes back to `\n`. To specify your own field-separator and newline characters (which must not occur in the data):

```
dsep='FN';  export dsep .
dsep='|\'  is the default.
```

C interface

d is only a few pages of C code, so it can be easily tailored or extended:

`#include "field.h"`
typedefs a struct `field`, which holds pointers to fieldname, fieldval etc.

`nf = getFields( char* rec, field f[ ])`
splits `rec` into 'fieldname: value' pairs, filling `f[ 0 .. nf-1]`.

`char* getPara( char nl )`
returns the next paragraph (lines up to `\n\n`) from the open file `ParaInfile`. The text is buffered inside `getPara`. Paragraphs may be no longer than 4096 bytes. `\n` s are changed to `nl`.

Possible extensions:

More complicated gets, such as `d get 'Duedate > 15.5'` .
Dates.
Iget, interactive get.

References

Knudsen D.B. et al, "A Modification Request Control System", in Proc. 2nd International Conference on Software Engineering, San Francisco, 1976, pp. 187-192.

Author

Denis Bzowy, April 1983

SEE ALSO

awk(1), cut(1), join(1), sort(1), dbm(3X)

BUGS

Sort, awk etc. may misbehave on records longer than 256 or 512 bytes.

NAME

date,time – Information about date and time

SYNOPSIS

character*10 day

character*8 clock

call date(day)

call time(clock)

DESCRIPTION

Date returns the day in the format dd.mm.yyyy in day.

Time returns the time in the format hh:mm:ss in clock.

NAME

dbminit, fetch, store, delete, firstkey, nextkey – data base subroutines

SYNOPSIS

typedef struct { char *dptr; int dsize; } datum;

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey();

datum nextkey(key);
datum key;

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two filesystem accesses. The functions are obtained with the loader option **–ldbm**.

Keys and *contents* are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has '.dir' as its suffix. The second file contains all data and has '.pag' as its suffix.

Before a database can be accessed, it must be opened by *dbminit*. At the time of this call, the files *file*.**dir** and *file*.**pag** must exist. (An empty database is created by creating zero-length '.dir' and '.pag' files.)

Once open, the data stored under a key is accessed by *fetch* and data is placed under a key by *store*. A key (and its associated contents) is deleted by *delete*. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of *firstkey* and *nextkey*. *Firstkey* will return the first key in the database. With any key *nextkey* will return the next key in the database. This code will traverse the data base:

```
for(key=firstkey(); key.dptr!=NULL; key=nextkey(key))
```

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates ok. Routines that return a *datum* indicate errors with a null (0) *dptr*.

BUGS

The '.pag' file will contain holes so that its apparent size is about four times its actual content. Older UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (cp, cat, tp, tar, ar) without filling in the holes.

Dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 512 bytes). Moreover all key/content pairs that hash together must fit on a single block. *Store* will return an error in the event that a disk block fills with inseparable data.

Delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *firstkey* and *nextkey* depends on a hashing function, not on anything interesting.

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

SYNOPSIS

```
double drand48 ( )

double erand48 (xsubi)
unsigned short xsubi[3];

long lrand48 ( )

long nrand48 (xsubi)
unsigned short xsubi[3];

long mrand48 ( )

long jrand48 (xsubi)
unsigned short xsubi[3];

void srand48 (seedval)
long seedval;

unsigned short *seed48 (seed16v)
unsigned short seed16v[3];

void lcong48 (param)
unsigned short param[7];
```

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions *drand48* and *erand48* return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

Functions *lrand48* and *nrand48* return non-negative long integers uniformly distributed over the interval $[0, 2^{31})$.

Functions *mrand48* and *jrand48* return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$.

Functions *srand48*, *seed48* and *lcong48* are initialization entry points, one of which should be invoked before either *drand48*, *lrand48* or *mrand48* is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if *drand48*, *lrand48* or *mrand48* is called without a prior call to an initialization entry point.) Functions *erand48*, *nrand48* and *jrand48* do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula

$$X_{n+1} = (aX_n + c)_{\mathrm{mod}\ m} \qquad n \geq 0.$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless *lcong48* has been invoked, the multiplier value a and the addend value c are given by

$$a = 5DEECE66D_{16} = 273673163155_8$$
$$c = B_{16} = 13_8.$$

The value returned by any of the functions *drand48*, *erand48*, *lrand48*, *nrand48*, *mrand48* or *jrand48* is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

The functions *drand48*, *lrand48* and *mrand48* store the last 48-bit X_i generated in an internal buffer; that is why they must be initialized prior to being invoked. The functions *erand48*, *nrand48* and *jrand48* require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions *erand48*, *nrand48* and *jrand48* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, i.e., the sequence of numbers in each stream will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function *srand48* sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function *seed48* sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by *seed48*, and a pointer to this buffer is the value returned by *seed48*. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last X_i value, and then use this value to reinitialize via *seed48* when the program is restarted.

The initialization function *lcong48* allows the user to specify the initial X_i, the multiplier value a, and the addend value c. Argument array elements *param*[*0-2*] specify X_i, *param*[*3-5*] specify the multiplier a, and *param*[*6*] specifies the 16-bit addend c. After *lcong48* has been called, a subsequent call to either *srand48* or *seed48* will restore the "standard" multiplier and addend values, a and c, specified on the previous page.

NOTES

The versions of these routines for the VAX-11 and PDP-11 are coded in assembly language for maximum speed. It requires approximately 80 μsec on a VAX-11/780 and 130 μsec on a PDP-11/70 to generate one pseudo-random number. On other computers, the routines are coded in portable C. The source code for the portable version can even be used on computers which do not have floating-point arithmetic. In such a situation, functions *drand48* and *erand48* do not exist; instead, they are replaced by the two new functions below.

```
long irand48 (m)
unsigned short m;

long krand48 (xsubi, m)
unsigned short xsubi[3], m;
```

Functions *irand48* and *krand48* return non-negative long integers uniformly distributed over the interval [0, *m*−1].

SEE ALSO

rand(3C).

NAME

ecvt, fcvt – output conversion

SYNOPSIS

char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
char *buf;

DESCRIPTION

Ecvt converts the *value* to a null-terminated string of *ndigit* ASCII digits and returns a pointer thereto. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero. The low-order digit is rounded.

Fcvt is identical to *ecvt*, except that the correct digit has been rounded for Fortran F-format output of the number of digits specified by **_ndigits*.

Gcvt converts the *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf*. It attempts to produce *ndigit* significant digits in Fortran F format if possible, otherwise E format, ready for printing. Trailing zeros may be suppressed.

SEE ALSO

printf(3S).

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

end, etext, edata – last locations in program

SYNOPSIS

extern end;
extern etext;
extern edata;

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of *etext* is the first address above the program text, *edata* above the initialized data region, and *end* above the uninitialized data region.

When execution begins, the program break coincides with *end*, but the program break may be reset by the routines of *brk*(2), *malloc*(3C), standard input/output (*stdio*(3S)), the profile (**−p**) option of *cc*(1), and so on. Thus, the current value of the program break should be determined by "sbrk(0)" (see *brk*(2)).

These symbols are accessible from assembly language if it is remembered that they should be prefixed by _.

SEE ALSO

brk(2), malloc(3C).

NAME

erf, erfc, derf, derfc — error function and complementary error function

SYNOPSIS

function erf(x)
real x

function erfc(x)
real x

double precisionc function derf(x)
double precision x

double precision function derfc(x)
double precision x

DESCRIPTION

Erf/derf returns the error function of *x*, defined as {2 over sqrt pi} int from 0 to x e sup {- t sup 2} dt .

Erfc/derfc, which returns 1.0 − *erf(x)*, is provided because of the extreme loss of relative accuracy if *erf(x)* is called for large *x* and the result subtracted from 1.0 (e.g. for *x* = 5, 12 places are lost).

SEE ALSO

exp(3M), erf(3M).

NAME

erf, erfc − error function and complementary error function

SYNOPSIS

```
#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;
```

DESCRIPTION

Erf returns the error function of *x*, defined as $\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$.

Erfc, which returns 1.0 − *erf(x)*, is provided because of the extreme loss of relative accuracy if *erf(x)* is called for large *x* and the result subtracted from 1.0 (e.g. for *x* = 5, 12 places are lost).

SEE ALSO

exp(3M).

NAME

exp, dexp, cexp – Fortran exponential intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2 = exp(r1)

dp2 = dexp(dp1)
dp2 = exp(dp1)

cx2 = clog(cx1)
cx2 = exp(cx1)
```

DESCRIPTION

Exp returns the real exponential function e^x of its real argument. *Dexp* returns the double-precision exponential function of its double-precision argument. *Cexp* returns the complex exponential function of its complex argument. The generic function *exp* becomes a call to *dexp* or *cexp* as required, depending on the type of its argument.

SEE ALSO

exp(3M).

NAME

exp, log, log10, pow, sqrt − exponential, logarithm, power, square root functions

SYNOPSIS

```
#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x;

double pow (x, y)
double x, y;

double sqrt (x)
double x;
```

DESCRIPTION

Exp returns e^x.

Log returns the natural logarithm of *x*. The value of *x* must be positive.

Log10 returns the logarithm base ten of *x*. The value of *x* must be positive.

Pow returns x^y. The values of *x* and *y* may not both be zero. If *x* is non-positive, *y* must be an integer.

Sqrt returns the square root of *x*. The value of *x* may not be negative.

DIAGNOSTICS

Exp returns HUGE when the correct value would overflow, and sets *errno* to ERANGE.

Log and *log10* return 0 and set *errno* to **EDOM** when *x* is non-positive. An error message is printed on the standard error output.

Pow returns 0 and sets *errno* to **EDOM** when *x* is non-positive and *y* is not an integer, or when *x* and *y* are both zero. In these cases a message indicating DOMAIN error is printed on the standard error output. When the correct value for *pow* would overflow, *pow* returns HUGE and sets *errno* to ERANGE.

Sqrt returns 0 and sets *errno* to **EDOM** when *x* is negative. A message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

hypot(3M), matherr(3M), sinh(3M).

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

```
#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;
```

DESCRIPTION

Fclose causes any buffers for the named *stream* to be emptied, and the file to be closed. Buffers allocated by the standard input/output system are freed.

Fclose is performed automatically upon calling *exit*(2).

Fflush causes any buffered data for the named output *stream* to be written to that file. The stream remains open.

These functions return 0 for success, and **EOF** if any errors were detected.

SEE ALSO

close(2), fopen(3S), setbuf(3S).

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

```
#include <stdio.h>

int feof (stream)
FILE *stream;

int ferror (stream)
FILE *stream

clearerr (stream)
FILE *stream

fileno(stream)
FILE *stream;
```

DESCRIPTION

Feof returns non-zero when end of file is read on the named input *stream*, otherwise zero.

Ferror returns non-zero when error has occurred reading or writing the named *stream*, otherwise zero. Unless cleared by *clearerr*, the error indication lasts until the stream is closed.

Clearerr resets the error indication on the named *stream*.

Fileno returns the integer file descriptor associated with the *stream*, see *open*(2).

Feof, *ferror*, and *fileno* are implemented as macros; they cannot be redeclared.

SEE ALSO

open(2), fopen(3S).

NAME

floor, ceil, fmod, fabs – floor, ceiling, remainder, absolute value functions

SYNOPSIS

```
#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

double fmod (x, y)
double x, y;

double fabs (x)
double x;
```

DESCRIPTION

Floor returns the largest integer (as a double-precision number) not greater than x.

Ceil returns the smallest integer not less than x.

Fmod returns x if y is zero, otherwise the number f with the same sign as x, such that $x = iy + f$ for some integer i, and $|f| < |y|$.

Fabs returns $|x|$.

SEE ALSO

abs(3C).

NAME

fopen, freopen, fdopen — open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen (file-name, type)
char *file-name, *type;

FILE *freopen (file-name, type, stream)
char *file-name, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;
```

DESCRIPTION

Fopen opens the file named by *file-name* and associates a stream with it. *Fopen* returns a pointer to be used to identify the stream in subsequent operations.

Type is a character string having one of the following values:

"r"	open for reading
"w"	create for writing
"a"	append; open for writing at end of file, or create for writing
"r+"	open for update (reading and writing)
"w+"	create for update
"a+"	append; open or create for update at end of file

Freopen substitutes the named file in place of the open *stream*. It returns the original value of *stream*. The original stream is closed, regardless of whether the open ultimately succeeds.

Freopen is typically used to attach the preopened constant names **stdin**, **stdout**, and **stderr** to specified files.

Fdopen associates a stream with a file descriptor obtained from *open*, *dup*, *creat*, or *pipe*(2). The *type* of the stream must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation which encounters end of file.

SEE ALSO

open(2), fclose(3S).

DIAGNOSTICS

Fopen and *freopen* return the pointer **NULL** if *file-name* cannot be accessed.

NAME

fp – Floating Point on the CADMUS

DESCRIPTION

There are currently three types of floating point (FP) available: the Motorola Fast Floating Point Package (FFP), the Motorola IEEE package, and FP hardware with the National Semiconductor FP coprocessor (NS).

Code generators for FFP are available in C, Fortran and Pascal, code generators for Motorola and NSC IEEE in C and Fortran. For both C and Fortran, the compiler option

-f	will produce code for FFP,
-fF	for Motorola IEEE, and
-fN	for NSC IEEE.

The options will influence the preprocessor, the compiler passes and the calling sequence of the loader. The shorthand

```
cc -f xxx.c
```

is expanded inside cc (resp. f77) to

```
cc -c -f -DFFP xxx.c; cc xxx.o -lffp .
```

```
cc -fF xxx.c
```

is expanded to

```
cc -c -fF -DIEEE -DMOT_IEEE xxx.c; cc xxx.o -lmot .
```

```
cc -fN xxx.c
```

is expanded to

```
cc -c -f -DIEEE -DNSC_IEEE xxx.c; cc xxx.o -lnsc .
```

The libraries libffp.a, libmot.a and libnsc.a contain code for these types of floating point, the stdio-routines for reading and printing floating-point values, and the mathematical routines formerly in libm.a. The library libm.a is no longer supported. The libraries exist in the 2 and 4 byte integer form, i.e. libffp.a and libLffp.a. The user need not be concerned about this. All he has to do is to specify the type of floating point he wishes, with -f, -fF or -fN, and if he wants 4 byte integers, with option -4.

Each program will call a routine fp_init() before it calls main(). The routine fp_init exists in each of the floating point libraries and in the standard library. The routine fp_init() in the standard library does nothing, whereas the routines in the FP libraries execute tasks specific for this kind of FP. If a program does not use floating point, then the routine in the standard library will be called, doing nothing. If a program uses floating point, then the corresponding FP library will be loaded before the standard library, and so the special fp_init routine will be called.

FFP

FFP is only available for 32 bit floating point numbers. The accuracy is about 7 1/3 digits. In C and Fortran double (resp. double precision) will be silently handled as float (real). FFP is a very fast software emulation, but fails if high accuracy or a large range is required.

Motorola IEEE (MOTIEEE)

MOTIEEE supports both single and double precision. Code is generated as if the instructions fmove, fadd, fmul etc. were part of the 68000 instruction set. When executed, these instructions will lead to an unimplemented instruction exception. The exception will be caught and an FP emulator invoked. In the case of the future Motorola FP coprocessor, the instruction will be executed directly by the coprocessor.

Unfortunately, the software emulation is quite slow. This is due to the emulation overhead and the complexities of the IEEE model. The advantage is that the switch to the Motorola FP coprocessor should be straight forward.

The MOTIEEE code emulates 8 floating point registers of extended precision (80 bits), plus some control registers. The C-compiler makes three FP registers available for register variables, thus the declaration

```
        register double d1,d2,d3;
```

makes sense. The control registers allow the specification of rounding modes and the enabling/disabling of floating point traps. Each program executes at the beginning the routine fp_init(). This routine sets the rounding mode to round-towards-zero, and enables traps for invalid operation, overflow, and divide-by-zero. Several routines are available for the user to read/write the control registers. If a trap occurs, the program will be signalled with signal SIGFPE. The precise cause of the exception can then be inquired by reading the FP status register.

The available special IEEE routines are:

```
ieee_get_cntrl()            /* return the contents of CNTRL register */

ieee_put_cntrl(mode)        /* write mode into CNTRL register */
int mode;

ieee_get_status()           /* return the contents of STATUS register */

ieee_put_status(stat)       /* write stat into STATUS register */
int stat;

ieee_get_tmpstat()          /* return the contents of TEMPSTAT register */

ieee_isnan(d)               /* return 1 if d is a NaN, 0 otherwise */
double d;

ieee_isinf(d)               /* return 1 if d is infinity, 0 otherwise */
double d;

ieee_isnorm(d)              /* return 1 if d is normalized, 0 otherwise */
double d;
```

The following is a list of /usr/include/mot.ieee.h. It explains the contents and use of the CNTRL, STATUS and TEMPSTAT registers mentioned above.

```
/* defines for the Motorola IEEE package */

/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */

/* Status Register (STATUS) */
    15     14     13     12     11     10     9     8
  +-----------------------------------------------------------+
| xxxx | xxxx |        IOP_CODE         |   CMPCC   |
+-------------------------------------------------+

    7      6      5      4      3      2     1     0
+-------------------------------------------------+
| xxxx | xxxx | IOVF | INEX |  DZ  | UNFL | OVFL | IOP |
+-------------------------------------------------+

#define IOP_CODE 0x3c00 /* set if IOP set */
#define CMPCC    0x0300 /* condition codes after fcmp */
#define IOVF     0x0020 /* integer overflow */
#define INEX     0x0010 /* inexact result */
#define DZ       0x0008 /* divide by zero */
#define UNFL     0x0004 /* Underflow */
#define OVFL     0x0002 /* Overflow */
#define IOP      0x0001 /* invalid operation */

/*
values for IOP_CODE:
0x0000  No IOP error
0x0040  Square root of a negative number, infinity in projective mode,
        or a not normalized number
0x0080  (+infinity) + (-infinity) in affine mode
0x00c0  Tried to convert NaN to binary integer
0x0100  In division: 0/0, infinity/infinity or divisor
        is not normalized and the dividend is not zero
        and is finite
0x0140  One of the input arguments was a trapping NaN
0x0180  Unordered condition tested by predicate other than
        equal or not-equal
0x01c0  Projective closure use of +/- infinity
0x0200  0 * infinity
0x0240  in REM <ea> is zero or not normalized or FPn
        is infinity
0x0280  Value of 'k' for BINDEC or 'p' for DECBIN is out
        of range
0x02c0  Tried to MOV a single denormalized number to a
        double destination
0x0300  Tried to return an unnormalized number to single
        or double (invalid result)
0x0340  Illegal instruction
0x0380  unused
0x03c0  unused

values for CMPCC:
#define 0x0000  equal
#define 0x0100  less than
#define 0x0200  greater than
#define 0x0300  unordered

The bits IOVF to IOP in the status register are set if any
errors have occurred. Note that each bit of these bits must be reset
by the caller. The FP processor only writes 1 bits and never clears
existing bits. This is done so a long computation can be completed
with the error status checked once at the end.
*/

/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Temporary trap status byte (TEMPSTAT) */

/*
Same defines as for IOVF to IOP  above.  The  bits  in  the  temporary
status byte represent the status returned from the last floating-point
operation.  They are cleared at the start of each operation. Due to an
apparent error in the Emulator, reading this byte (or STATUS or CNTRL)
after a trap took place takes another trap of the same kind. Disabling
the traps  before  reading  TEMPSTAT  is  an  operation  which  clears
TEMPSTAT!  So the only use of TEMPSTAT currently lies in disabling all
traps and then checking TEMPSTAT after suspicious operations.
```

```
*/

/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Control register (CNTRL) */
   15      14      13      12      11      10      9       8
+--------------------------------------------------------------+
| xxxx | xxxx |     PREC      | NORM |CLOSUR|     ROUND        |
+--------------------------------------------------------------+

    7      6       5      4       3       2       1       0
+--------------------------------------------------------------+
| xxxx | xxxx | IOVF | INEX |  DZ   | UNFL | OVFL |   IOP |
+--------------------------------------------------------------+

#define PREC     0x3000
#define NORM     0x0800
#define CLOSURE  0x0400
#define ROUND    0x0300
/* IOVF to IOP as above */

/*
values for PREC:
0x0000  round to extended
0x1000  round to single
0x2000  round to double
0x3000  unused, reserved

note: C stdio and Fortran demand round-to-zero !!!

values for NORM:
0x0000  do not normalize denormalized operands before an
        operation (warning mode)
0x0800  normalize denormalized numbers while converting
        to internal format (normalizing mode)

note: Unnormalized numbers are not affected by bit NORM


values for CLOSURE:
0x0000  projective closure
0x0400  affine closure

values for ROUND:
0x0000  round to nearest
0x0100  round to zero
0x0200  round to plus infinity
0x0300  round to plus infinity

IOVF to IOP:
The programmer may set a one in any bit to enable a trap
on the corresponding error conditiom
*/
```

The procedure fp_init executes (among others) the statement

```
        ieee_put_cntrl(0x010b);
```

thus setting rounding mode to round-to-zero, and enabling the traps for DZ, OVFL and IOP. If any of these occur, signal SIGFPE will be sent.

National Semiconductor IEEE (NSCIEEE)

This FP code runs with the Cadmus FP board, which contains the NSC 16081 FP coprocessor. This chip is seen from the code generator as containing 4 double precision registers. This is not sufficient to allow register variables. During a context switch, the state of the NSC must be saved. But it would be wasteful to save the context if the process does not use the NSC at all. So fp_init() will execute a system call, that tells the kernel to save or restore the context when this process loses or regains the CPU.

The NSC processor is seen as a piece of memory, 64kb large, accessible in user memory. FP operations are executed by loading and storing special addresses in this memory. If the user accesses this memory by himself, anything can happen.

The procedures (sic)

```
long sfsr();   and
lfsr(a); long a;
```

are available to load and store the NSC status register.

COMPATIBILITY

It is not permitted to load together modules which have been compiled for different floating point codes. However, programs compiled for Motorola or NSC IEEE should behave identically. The compiler use internally for constant expression evaluation FFP and MOTIEEE only, as the existence of a NSC board can not be guaranteed.

Motorola distributes the IEEE package with the claim that the code can be executed by hardware, when the coprocessor is attached. However it is clear from the specifications of the 68020 and the 68881, that the code generated for the software emulation is not compatible with the coprocessor! The instruction format is different and many instructions are missing from the software emulation. Still, the software emulation package is 15kb large! So it can be forseen that even a fourth kind of FP will be necessary for the 68881, unless Motorola provides software that exactly emulates the 68881.

NAME

fread, fwrite – buffered binary input/output

SYNOPSIS

```
#include <stdio.h>

int fread ((char *) ptr, sizeof (*ptr), nitems, stream)
FILE *stream;

int fwrite ((char *) ptr, sizeof (*ptr), nitems, stream)
FILE *stream;
```

DESCRIPTION

Fread reads, into a block beginning at *ptr*, *nitems* of data of the type of **ptr* from the named input *stream*. It returns the number of items actually read.

Fwrite appends at most *nitems* of data of the type of **ptr* beginning at *ptr* to the named output *stream*. It returns the number of items actually written.

SEE ALSO

read(2), write(2), fopen(3S), getc(3S), putc(3S), gets(3S), puts(3S), printf(3S), scanf(3S).

NAME

frexp, ldexp, modf – split into mantissa and exponent

SYNOPSIS

```
double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;

double modf (value, iptr)
double value, *iptr;
```

DESCRIPTION

Frexp returns the mantissa of a double *value* as a double quantity, *x*, of magnitude less than 1 and stores an integer *n* such that *value* = *x**2***n* indirectly through *eptr*.

Ldexp returns the quantity *value**2***exp*.

Modf returns the positive fractional part of *value* and stores the integer part indirectly through *iptr*.

NAME

fseek, ftell, rewind – reposition a stream

SYNOPSIS

```
#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

long ftell (stream)
FILE *stream;

rewind(stream)
FILE *stream;
```

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, according as *ptrname* has the value 0, 1, or 2.

Fseek undoes any effects of *ungetc*(3S).

After *fseek* or *rewind*, the next operation on an update file may be either input or output.

Ftell returns the current value of the offset relative to the beginning of the file associated with the named *stream*. The offset is measured in bytes on UNIX 3.0 and UNIX/RT; on some other systems, it is a magic cookie and is the only foolproof way to obtain an *offset* for *fseek*.

Rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, 0).

SEE ALSO

lseek(2), fopen(3S).

DIAGNOSTICS

Fseek returns non-zero for improper seeks, otherwise zero.

NAME

ftw – walk a file tree

SYNOPSIS

```
#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn) ( );
int depth;
```

DESCRIPTION

Ftw recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure (see *stat*(2)) containg information about the object, and an integer. Possible values of the integer, defined in the <ftw.h> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which *stat* could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

Ftw visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some error is detected within *ftw* (such as an I/O error). If the tree is exhausted, *ftw* returns zero. If *fn* returns a nonzero value, *ftw* stops its tree traversal and returns whatever value was returned by *fn*. If *ftw* detects an error, it returns –1, and sets the error type in *errno*.

Ftw uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero or negative, the effect is the same as if it were 1. *Depth* must not be greater than the number of file descriptors currently available for use. *Ftw* will run more quickly if *depth* is at least as large as the number of levels in the tree.

SEE ALSO

stat(2), malloc(3C).

BUGS

Because *ftw* is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.
It could be made to run faster and use less storage on deep structures at the cost of considerable complexity.
Ftw uses *malloc*(3C) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

NAME

int, ifix, idint, real, float, sngl, dble, cmplx, dcmplx, ichar, char – explicit Fortran type conversion

SYNOPSIS

```
integer i, j
real r, s
double precision dp, dq
complex cx
double complex dcx
character*1 ch

i = int(r)
i = int(dp)
i = int(cx)
i = int(dcx)
i = ifix(r)
i = idint(dp)

r = real(i)
r = real(dp)
r = real(cx)
r = real(dcx)
r = float(i)
r = sngl(dp)

dp = dble(i)
dp = dble(r)
dp = dble(cx)
dp = dble(dcx)

cx = cmplx(i)
cx = cmplx(i, j)
cx = cmplx(r)
cx = cmplx(r, s)
cx = cmplx(dp)
cx = cmplx(dp, dq)
cx = cmplx(dcx)

dcx = dcmplx(i)
dcx = dcmplx(i, j)
dcx = dcmplx(r)
dcx = dcmplx(r, s)
dcx = dcmplx(dp)
dcx = dcmplx(dp, dq)
dcx = dcmplx(cx)

i = ichar(ch)
ch = char(i)
```

DESCRIPTION

These functions perform conversion from one data type to another.

int converts to *integer* form its *real*, *double precision*, *complex*, or *double complex* argument. If the argument is *real* or *double precision*, int returns the integer whose magnitude is the largest integer that does not exceed the magnitude of the argument and whose sign is the same as the sign of the argument (i.e. truncation). For complex types, the above rule is applied to the real part. **ifix** and **idint** convert only *real* and *double precision* arguments respectively.

real converts to *real* form an *integer*, *double precision*, *complex*, or *double complex* argument. If the argument is *double precision* or *double complex*, as much precision is kept as is possible. If the argument is one of the complex types, the real part is returned. **float** and **sngl** convert only *integer* and *double precision* arguments respectively.

dble converts any *integer*, *real*, *complex*, or *double complex* argument to *double precision* form. If the argument is of a complex type, the real part is returned.

cmplx converts its *integer*, *real*, *double precision*, or *double complex* argument(s) to *complex* form.

dcmplx converts to *double complex* form its *integer*, *real*, *double precision*, or *complex* argument(s).

Either one or two arguments may be supplied to **cmplx** and **dcmplx** . If there is only one argument, it is taken as the real part of the complex type and a imaginary part of zero is supplied. If two arguments are supplied, the first is taken as the real part and the second as the imaginary part.

ichar converts from a character to an integer depending on the character's position in the collating sequence.

char returns the character in the *i*th position in the processor collating sequence where *i* is the supplied argument.

For a processor capable of representing *n* characters,

ichar(**char**(i)) = i for 0 <= i < *n*, and

char(**ichar**(ch)) = ch for any representable character *ch*.

NAME

gamma – log gamma function

SYNOPSIS

```
#include <math.h>

extern int signgam;

double gamma (x)
double x;
```

DESCRIPTION

Gamma returns $\ln(|\Gamma(x)|)$, where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t}t^{x-1}dt$. The sign of $\Gamma(x)$ is returned in the external integer *signgam*. The argument x may not be a non-positive integer.

The following C program fragment might be used to calculate Γ:

```
if ((y = gamma(x)) > LOGHUGE)
        error();
y = signgam * exp(y);
```

where LOGHUGE is the least value that causes *exp*(3M) to return a range error.

DIAGNOSTICS

For non-negative integer arguments **HUGE** is returned, and *errno* is set to **EDOM**. A message indicating DOMAIN error is printed on the standard error output.

If the correct value would overflow, *gamma* returns **HUGE** and sets *errno* to **ERANGE**.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

exp(3M), matherr(3M).

NAME

getarg, getar2 – return Fortran command-line argument

SYNOPSIS

character*N c
integer*4 i
integer*2 j

getarg(i, c)

getar2(j, c)

DESCRIPTION

Getarg resp. getar2 returns the ***i***-th command-line argument of the current process. Thus, if a program were invoked via

```
foo arg1 arg2 arg3
```

getarg(2, **c***) resp. getar2(2,* **c***)* would return the string "arg2" in the character variable **c**,ifcis*long*enough. the length of c.

SEE ALSO

getopt(3C).

NAME

getc, getchar, fgetc, getw – get character or word from stream

SYNOPSIS

#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ()

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;

DESCRIPTION

Getc returns the next character from the named input *stream*.

Getchar( ) is identical to *getc*(*stdin*).

Fgetc behaves like *getc*, but is a genuine function, not a macro; it may therefore be used as an argument. *Fgetc* runs more slowly than *getc*, but takes less space per invocation.

Getw returns the next word from the named input *stream*. It returns the constant **EOF** upon end of file or error, but since that is a valid integer value, *feof* and *ferror*(3S) should be used to check the success of *getw*. *Getw* assumes no special alignment in the file.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S).

DIAGNOSTICS

These functions return the integer constant **EOF** at end of file or upon read error.

A stop with message "Reading bad file" means that an attempt has been made to read from a stream that has not been opened for reading by *fopen*.

BUGS

Getc and its variant *getchar* return **EOF** on end of file; this is wiser than, but incompatible with, the older *getchar*(3S).
Because it is implemented as a macro, *getc* treats incorrectly a *stream* argument with side effects. In particular, **getc(*f++);** doesn't work sensibly.

NAME

getcwd – get path-name of current working directory

SYNOPSIS

char *getcwd (buf, size)
char *buf;
int size;

DESCRIPTION

Getcwd returns a pointer to the current directory path-name. The value of *size* must be at least two greater than the length of the path-name to be returned.

If *buf* is a NULL pointer, *getcwd* will obtain *size* bytes of space using *malloc*(3C). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free*.

The function is implemented by using *popen*(3S) to pipe the output of the *pwd*(1) command into the specified string space.

EXAMPLE

```
char *cwd, *getcwd();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
        perror("pwd");
        exit(1);
}
printf("%s\n", cwd);
```

SEE ALSO

pwd(1), malloc(3C), popen(3S).

DIAGNOSTICS

Returns NULL with *errno* set if *size* is not large enough, or if an error ocurrs in a lower-level function.

NAME

getenv − value for environment name

SYNOPSIS

```
char *getenv (name)
char *name;
```

DESCRIPTION

Getenv searches the environment list (see *environ*(7)) for a string of the form *name=value* and returns *value* if such a string is present, otherwise 0 (NULL).

SEE ALSO

environ(7).

NAME

getenv – return Fortran environment variable

SYNOPSIS

character*N c

getenv("TMPDIR", c)

DESCRIPTION

Getenv returns the character-string value of the environment variable represented by its first argument into the character variable of its second argument. If no such environment variable exists, all blanks will be returned.

SEE ALSO

getenv(3C), environ(7).

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent – get group file entry

SYNOPSIS

#include <grp.h>

struct group *getgrent ( );

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

int setgrent ( );

int endgrent ( );

DESCRIPTION

Getgrent, *getgrgid* and *getgrnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the group file.

```
struct    group { /* see getgrent(3) */
        char*gr_name;
        char*gr_passwd;
        int  gr_gid;
        char**gr_mem;
};

struct group *getgrent();
struct group *getgrgid();
struct group *getgrnam();
```

The members of this structure are:

gr_name	The name of the group.
gr_passwd	The encrypted password of the group.
gr_gid	The numerical group ID.
gr_mem	Null-terminated vector of pointers to the individual member names.

Getgrent reads the next line of the file, so successive calls may be used to search the entire file. *Getgrgid* and *getgrnam* search from the beginning of the file until a matching *gid* or *name* is found, or **EOF** is encountered.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *Endgrent* may be called to close the group file when processing is complete.

FILES

/etc/group

SEE ALSO

getlogin(3C), getpwent(3C), group(5).

DIAGNOSTICS

A null pointer (0) is returned on **EOF** or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getlogin – get login name

SYNOPSIS

char *getlogin ( );

DESCRIPTION

Getlogin returns a pointer to the login name as found in **/etc/utmp**. It may be used in conjunction with *getpwnam* to locate the correct password file entry when the same user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a typewriter, it returns **NULL**. The correct procedure for determining the login name is to call *cuserid*, or to call *getlogin* and if it fails, to call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(5).

DIAGNOSTICS

Returns **NULL** if name not found.

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

getopt – get option letter from argv

SYNOPSIS

int getopt (argc, argv, optstring)
int argc;
char **argv;
char *optstring;
extern char *optarg;
extern int optind;

DESCRIPTION

Getopt returns the next option letter in *argv* that matches a letter in *optstring*. *Optstring* is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. *Optarg* is set to point to the start of the option argument on return from *getopt*.

Getopt places in *optind* the *argv* index of the next argument to be processed. Because *optind* is external, it is normally initialized to zero automatically before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns **EOF**. The special option — may be used to delimit the end of the options; **EOF** will be returned, and — will be skipped.

DIAGNOSTICS

Getopt prints an error message on *stderr* and returns a question mark (?) when it encounters an option letter not included in *optstring*.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the options **f** and **o**, both of which require arguments:

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern int optind;
        extern char *optarg;
        .
        .
        .
        while ((c = getopt (argc, argv, "abf:o:")) != EOF)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
                        bproc();
                    break;
                case 'f':
                    ifile = optarg;
                    break;
                case 'o':
                    ofile = optarg;
                    bufsiza = 512;
                    break;
                case '?':
                    errflg++;
                }
            if (errflg) {
                fprintf (stderr, "usage: . . . ");
                exit (2);
            }
            for( ; optind < argc; optind++) {
                if (access (argv[optind], 4)) {
            :
            :
        }
```

NAME

getpass – read a password

SYNOPSIS

char *getpass (prompt)
char *prompt;

DESCRIPTION

Getpass reads a password from the file **/dev/tty**, or if that cannot be opened, from the standard input, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters.

FILES

/dev/tty

SEE ALSO

crypt(3C).

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpw – get name from UID

SYNOPSIS

getpw (uid, buf)
int uid;
char *buf;

DESCRIPTION

Getpw searches the password file for the (numerical) ***uid***, and fills in *buf* with the corresponding line; it returns non-zero if ***uid*** could not be found. The line is null-terminated.

This routine is included only for compatibility with prior systems and should not be used; see *getpwent*(3C) for routines to use instead.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(5).

DIAGNOSTICS

Non-zero return on error.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent – get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent ( );

struct passwd *getpwuid (uid)

int uid;

struct passwd *getpwnam (name)

char *name;

int setpwent ( );

int endpwent ( );
```

DESCRIPTION

Getpwent, *getpwuid* and *getpwnam* each returns a pointer to an object with the following structure containing the broken-out fields of a line in the password file.

```
struct  passwd
{
        char    *pw_name;
        char    *pw_passwd;
        int     pw_uid;
        int     pw_gid;
        char    *pw_age;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};

struct comment {
        char    *c_dept;
        char    *c_name;
        char    *c_acct;
        char    *c_bin;
};

struct passwd *getpwent();
struct passwd *getpwuid();
struct passwd *getpwnam();
```

The *pw_comment* field is unused; the others have meanings described in *passwd*(5).

Getpwent reads the next line in the file, so successive calls can be used to search the entire file. *Getpwuid* and *getpwnam* search from the beginning of the file until a matching *uid* or *name* is found, or EOF is encountered.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *Endpwent* may be called to close the password file when processing is complete.

FILES

/etc/passwd

SEE ALSO

getlogin(3C), getgrent(3C), passwd(5).

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

gets, fgets – get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

DESCRIPTION

Gets reads a string into *s* from the standard input stream **stdin**. The string is terminated by a new-line character, which is replaced in *s* by a null character. *Gets* returns its argument.

Fgets reads *n*−1 characters, or up to a new-line character (which is retained), whichever comes first, from the *stream* into the string *s*. The last character read into *s* is followed by a null character. *Fgets* returns its first argument.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), puts(3S), scanf(3S).

DIAGNOSTICS

Gets and *fgets* return the constant pointer **NULL** upon end-of-file or error.

NOTE

Gets deletes the new-line ending its input, but *fgets* keeps it.

NAME

getutent, getutid, getutline, pututline, setutent, endutent, utmpname – access utmp file entry

SYNOPSIS

```
#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid (id)
struct utmp *id;

struct utmp *getutline (line)
struct utmp *line;

void pututline (utmp)
struct utmp *utmp;

void setutent ( )

void endutent ( )

void utmpname (file)
char *file;
```

DESCRIPTION

Getutent, *getutid* and *getutline* each return a pointer to a structure of the following type:

```
struct  utmp {
        char    ut_user[8];      /* User login name */
        char    ut_id[4];        /* /etc/inittab id (usually line #) */
        char    ut_line[12];     /* device name (console, lnxx) */
        short   ut_pid;          /* process id */
        short   ut_type;         /* type of entry */
        struct  exit_status {
            short e_termination;         /* Process termination status */
            short e_exit;                /* Process exit status */
        } ut_exit;               /* The exit status of a process
                                  * marked as DEAD_PROCESS. */
        time_t  ut_time;         /* time entry was made */
};
```

Getutent reads in the next entry from a *utmp*-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

Getutid searches forward from the current point in the *utmp* file until it finds an entry with a *ut_type* matching *id->ut_type* if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME or NEW_TIME. If the type specified in *id* is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS or DEAD_PROCESS, then *getutid* will return a pointer to the first entry whose type is one of these four and whose *ut_id* field matches *id->ut_id*. If the end of file is reached without a match, it fails.

Getutline searches forward from the current point in the *utmp* file until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a *ut_line* string matching the *line->ut_line* string. If the end of file is reached without a match, it fails.

Pututline writes out the supplied *utmp* structure into the *utmp* file. It uses *getutid* to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of *pututline* will have searched for the proper entry using one of the *getut* routines. If so, *pututline* will not search. If *pututline* does not find a matching slot for the new entry, it will add a new entry to the end of the file.

Setutent resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

Endutent closes the currently open file.

Utmpname allows the user to change the name of the file examined, from /etc/utmp to any other file. It is most often expected that this other file will be /etc/wtmp. If the file doesn't exist, this will not be apparent until the first attempt to reference the file is made. *Utmpname* does not open the file. It just closes the old file if it is currently open and saves the new file name.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(5).

DIAGNOSTICS

A NULL pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

COMMENTS

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason to use *getutline* to search for multiple occurences, it would be necessary to zero out the static after each success, or *getutline* would just return the same pointer over and over again. There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututline* if it finds that it isn't already at the correct place in the file will not hurt the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, if the user has just modified those contents and passed the pointer back to *pututline*.

These routines use buffered standard I/O for input, but *pututline* uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

NAME

hsearch, hcreate, hdestroy – manage hash search tables

SYNOPSIS

```
#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ( )
```

DESCRIPTION

Hsearch is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *Item* is a structure of type ENTRY (defined in the <search.h> header file) containing two pointers: *item.key* points to the comparison key, and *item.data* points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) *Action* is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. **ENTER** indicates that the item should be inserted in the table at an appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

Hcreate allocates sufficient space for the table, and must be called before *hsearch* is used. *nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

Hdestroy destroys the search table, and may be followed by another call to *hcreate*.

NOTES

Hsearch uses *open addressing* with a *multiplicative* hash function. However, its source code has many other options available which the user may select by compiling the *hsearch* source with the following symbols defined to the preprocessor:

DIV Use the *remainder modulo table size* as the hash function instead of the multiplicative algorithm.

USCR Use a User Supplied Comparison Routine for ascertaining table membership. The routine should be named *hcompar* and should behave in a mannner similar to *strcmp* (see *string*(3C)).

CHAINED Use a linked list to resolve collisions. If this option is selected, the following other options become available.

- **START** Place new entries at the beginning of the linked list (default is at the end).
- **SORTUP** Keep the linked list sorted by key in ascending order.

SORTDOWN Keep the linked list sorted by key in descending order.

Additionally, there are preprocessor flags for obtaining debugging printout (−DDEBUG) and for including a test driver in the calling routine (−DDRIVER). The source code should be consulted for further details.

SEE ALSO
bsearch(3C), lsearch(3C), string(3C), tsearch(3C).

DIAGNOSTICS
Hsearch returns a NULL pointer if either the action is **FIND** and the item could not be found or the action is **ENTER** and the table is full.

Hcreate returns zero if it cannot allocate sufficient space for the table.

BUGS
Only one hash search table may be active at any given time.

NAME

hypot − Euclidean distance function

SYNOPSIS

```
#include <math.h>        ?

double hypot (x, y)
double x, y;
```

DESCRIPTION

Hypot returns

sqrt(x * x + y * y),

taking precautions against unwarranted overflows.

DIAGNOSTICS

When the correct value would overflow, *hypot* returns **HUGE** and sets *errno* to **ERANGE**.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M), sqrt(3F).

NAME

iargc – Number of command-line arguments

SYNOPSIS

integer i

i = iargc()

DESCRIPTION

Iargc returns the number of arguments in the command-line (zero for no arguments except the program-name). If a program were invoked via

```
foo arg1 arg2 arg3
```

iargc() would return 3.

SEE ALSO

getopt(3C), getarg(3f).

NAME

index – return location of Fortran substring

SYNOPSIS

character*N1 ch1
character*N2 ch2
integer i

i = index(ch1, ch2)

DESCRIPTION

Index returns the location of substring *ch2* in string *ch1*. The value returned is the position at which substring *ch2* starts, or 0 is it is not present in string *ch1*.

NAME

int2 – convert 4-Byte Integer to 2-Byte Integer

SYNOPSIS

integer*4 long
integer*2 short

short = int2(long)

DESCRIPTION

Int2 converts a 4-Byte Integer to 2-Byte Integer (truncation of the 2 MSBytes of the 4-Byte Integer).

NAME

int4 – convert 2-Byte Integer to 4-Byte Integer

SYNOPSIS

integer*4 long
integer*2 short

long = int4(short)

DESCRIPTION

Int4 converts a 2-Byte Integer to 4-Byte Integer.

NAME

l3tol, ltol3 – convert between 3-byte integers and long integers

SYNOPSIS

```
l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

DESCRIPTION

L3tol converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

Ltol3 performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance where the block numbers are three bytes long.

SEE ALSO

fs(5).

NAME

len – return length of Fortran string

SYNOPSIS

```
character*N ch
integer i

i = len(ch)
```

DESCRIPTION

Len returns the length of string *ch*.

NAME

log, alog, dlog, clog – Fortran natural logarithm intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2 = alog(r1)
r2 = log(r1)

dp2 = dlog(dp1)
dp2 = log(dp1)

cx2 = clog(cx1)
cx2 = log(cx1)
```

DESCRIPTION

Alog returns the real natural logarithm of its real argument. *Dlog* returns the double-precision natural logarithm of its double-precision argument. *Clog* returns the complex logarithm of its complex argument. The generic function *log* becomes a call to *alog*, *dlog*, or *clog* depending on the type of its argument.

SEE ALSO

exp(3M).

NAME

log10, alog10, dlog10 – Fortran common logarithm intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = alog10(r1)
r2 = log10(r1)

dp2 = dlog10(dp1)
dp2 = log10(dp1)

DESCRIPTION

Alog10 returns the real common logarithm of its real argument. *Dlog* returns the double-precision common logarithm of its double-precision argument. The generic function *log* becomes a call to *alog* or *dlog* depending on the type of its argument.

SEE ALSO

exp(3M).

NAME

logname – login name of user

SYNOPSIS

char *logname();

DESCRIPTION

Logname returns a pointer to the null-terminated login name; it extracts the **$LOGNAME** variable from the user's environment.

This routine is kept in /lib/libPW.a.

FILES

/etc/profile

SEE ALSO

env(1), login(1), profile(5), environ(7).

NAME

long – standard procedures modified for long arguments

SYNOPSIS

char *lmalloc (size) long size;

char *lrealloc (ptr, size)
char *ptr;
long size;

DESCRIPTION

Lmalloc and *lrealloc* are the same routines as *malloc* and *realloc* (see *malloc*(3c)) except that they have long arguments. They are only available in the 2 byte integer standard library.

NAME

lsearch – linear search and update

SYNOPSIS

char *lsearch ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );

DESCRIPTION

Lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. *Key* points to the datum to be sought in the table. *Base* points to the first element in the table. *Nelp* points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. *Compar* is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

bsearch(3C), hsearch(3C), tsearch(3C).

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

malloc, free, realloc, calloc — main memory allocator

SYNOPSIS

char *malloc (size) unsigned size;

free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned elem, elsize;

DESCRIPTION

Malloc and *free* provide a simple general-purpose memory allocation package. *Malloc* returns a pointer to a block of at least *size* bytes beginning on a word boundary.

The argument to *free* is a pointer to a block previously allocated by *malloc*; this space is made available for further allocation, but its contents are left undisturbed.

Needless to say, grave disorder will result if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

Malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls *sbrk* (see *brk*(2)) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

Realloc also works if *ptr* points to a block freed since the last call of *malloc*, *realloc*, or *calloc*; thus sequences of *free*, *malloc* and *realloc* can exploit the search strategy of *malloc* to do storage compaction.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

DIAGNOSTICS

Malloc, *realloc* and *calloc* return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When *realloc* returns 0, the block pointed to by *ptr* may be destroyed.

NAME

matherr – error-handling function

SYNOPSIS

```
#include <math.h>

int matherr (x)
struct exception *x;
```

DESCRIPTION

Matherr is invoked by functions in the Math Library when errors are detected. Users may define their own procedures for handling errors by including a function named *matherr* in their programs. *Matherr* must be of the form described above. A pointer to the exception structure *x* will be passed to the user-supplied *matherr* function when an error occurs. This structure, which is defined in the *<math.h>* header file, is as follows:

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
};
```

The element *type* is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

DOMAIN	domain error
SING	singularity
OVERFLOW	overflow
UNDERFLOW	underflow
TLOSS	total loss of significance
PLOSS	partial loss of significance

The element *name* points to a string containing the name of the function that had the error. The variables *arg1* and *arg2* are the arguments to the function that had the error. *Retval* is a double that is returned by the function having the error. If it supplies a return value, the user's *matherr* must return non-zero. If the default error value is to be returned, the user's *matherr* must return 0.

If *matherr* is not supplied by the user, the default error-handling procedures, described with the math functions involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, *errno* is set to non-zero and the program continues.

EXAMPLE

```
matherr(x)
register struct exception *x;
{
        switch (x->type) {
        case DOMAIN:
        case SING: /* print message and abort */
              fprintf(stderr, "domain error in %s\n", x->name);
              abort( );
        case OVERFLOW:
              if (!strcmp("exp", x->name)) {
                    /* if exp, print message, return the argument */
```

```
                fprintf(stderr, "exp of %f\n", x->arg1);
                x->retval = x->arg1;
            } else if (!strcmp("sinh", x->name)) {
                /* if sinh, set errno, return 0 */
                errno = ERANGE;
                x->retval = 0;
            } else
                /* otherwise, return HUGE */
                x->retval = HUGE;
            break;
        case UNDERFLOW:
            return (0); /* execute default procedure */
        case TLOSS:
        case PLOSS:
            /* print message and return 0 */
            fprintf(stderr, "loss of significance in %s\n", x->name);
            x->retval = 0;
            break;
        }
        return (1);
    }
```

DEFAULT ERROR HANDLING PROCEDURES						
	Types of Errors					
	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
BESSEL: y0, y1, yn (neg. no.)	– M, –H	– –	H –	0 –	– –	• –
EXP:	–	–	H	0	–	
POW: (neg.)**(non-int.), 0**0	– M, 0	– –	H –	0 –	– –	– –

LOG:
log(0):
log(neg.):
	–
	M, –H
		M, –H
		–
			–
			–
				–
				–
					–
					–
						–
						–

SQRT:	M, 0	–	–	–	–	–
GAMMA:	–	M, H	–	–	–	–
HYPOT:	–	–	H	–	–	–
SINH, COSH:	–	–	H	–	–	–
SIN, COS:	–	–	–	–	M, 0	M, •
TAN:	–	–	H	–	0	•
ACOS, ASIN:	M, 0	–	–	–	–	–

ABBREVIATIONS	
•	As much as possible of the value is returned.
M	Message is printed.
H	HUGE is returned.
–H	–HUGE is returned.
0	0 is returned.

NAME

max, max0, amax0, max1, amax1, dmax1 – Fortran maximum-value functions

SYNOPSIS

```
integer i, j, k, l
real a, b, c, d
double precision dp1, dp2, dp3

l = max(i, j, k)
c = max(a, b)
dp = max(a, b, c)
k = max0(i, j)
a = amax0(i, j, k)
i = max1(a, b)
d = amax1(a, b, c)
dp3 = dmax1(dp1, dp2)
```

DESCRIPTION

The maximum-value functions return the largest of their arguments (of which there may be any number). *Max* is the generic form which can be used for all data types and takes its return type from that of its arguments (which must all be of the same type). *Max0* returns the integer form of the maximum value of its integer arguments; *amax0*, the real form of its integer arguments; *max1*, the integer form of its real arguments; *amax1*, the real form of its real arguments; and *dmax1*, the double-precision form of its double-precision arguments.

SEE ALSO

min(3F).

NAME

mclock – return Fortran time accounting

SYNOPSIS

integer i

i = mclock( )

DESCRIPTION

Mclock returns time accounting information about the current process and its child processes. The value returned is the sum of the current process's user time and the user and system times of all child processes.

SEE ALSO

times(2), clock(3C), system(3F).

NAME

min, min0, amin0, min1, amin1, dmin1 – Fortran minimum-value functions

SYNOPSIS

```
integer i, j, k, l
real a, b, c, d
double precision dp1, dp2, dp3

l = min(i, j, k)
c = min(a, b)
dp = min(a, b, c)
k = min0(i, j)
a = amin0(i, j, k)
i = min1(a, b)
d = amin1(a, b, c)
dp3 = dmin1(dp1, dp2)
```

DESCRIPTION

The minimum-value functions return the minimum of their arguments (of which there may be any number). *Min* is the generic form which can be used for all data types and takes its return type from that of its arguments (which must all be of the same type). *Min0* returns the integer form of the minimum value of its integer arguments; *amin0*, the real form of its integer arguments; *min1*, the integer form of its real arguments; *amin1*, the real form of its real arguments; and *dmin1*, the double-precision form of its double-precision arguments.

SEE ALSO

max(3F).

NAME

mktemp – make a unique file name

SYNOPSIS

char *mktemp (template)
char *template;

DESCRIPTION

Mktemp replaces *template* by a unique file name, and returns the address of the template. The template should look like a file name with six trailing **X**s, which will be replaced with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file.

SEE ALSO

getpid(2).

BUGS

It is possible to run out of letters.

NAME

mod, amod, dmod – Fortran remaindering intrinsic functions

SYNOPSIS

integer i, j, k
real r1, r2, r3
double precision dp1, dp2, dp3

k = mod(i, j)

r3 = amod(r1, r2)
r3 = mod(r1, r2)

dp3 = dmod(dp1, dp2)
dp3 = mod(dp1, dp2)

DESCRIPTION

Mod returns the integer remainder of its first argument divided by its second argument. *Amod* and *dmod* return, respectively, the real and double-precision whole number remainder of the integer division of their two arguments. The generic version *mod* will return the data type of its arguments.

NAME

monitor – prepare execution profile

SYNOPSIS

monitor (lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)( ), (*highpc)( );
short buffer[ ];
long bufsize;
int nfunc;

DESCRIPTION

An executable program created by **cc −p** automatically includes calls for *monitor* with default parameters; *monitor* needn't be called explicitly except to gain fine control over profiling.

Monitor is an interface to *profil*(2). *Lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* short integers. *Monitor* arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. At most *nfunc* call counts can be kept; only calls of functions compiled with the profiling option **−p** of *cc*(1) are recorded. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext(), _entry();
...
monitor(&_entry, etext, buf, bufsize, nfunc);
```

Etext lies just above all the program text, see *end*(3C).

To stop execution monitoring and write the results on the file **mon.out**, use

```
monitor((int *)0);
```

prof(1) can then be used to examine the results.

FILES

mon.out

SEE ALSO

cc(1), prof(1), profil(2).

NAME

nfcomment – a user interface to the notesfile system

SYNOPSIS

nfcomment (notesfile, text, title, dirflag, anonflag)
char *nfname, *text, *title;

DESCRIPTION

nfcomment provides user programs with the ability to insert *notes* into a *notesfile*.

The note is inserted into the notesfile specified by *nfname*. *Text* is the address of the body of the note; this must be null-terminated. If *text* is NULL, the note is gathered from standard input until an EOF is encountered. The note is entered with the title specified by the *title* parameter. If the *dirflag* or *anonflag* parameters are non-zero, the director message is enabled or the note is entered anonymously. These take effect only if the user has the appropriate priviledges in the notesfile.

Nfpipe is used to make the actual insertion of the text.

FILES

/usr/lib/libnfcom.a	-lnfcom library

SEE ALSO

notes(1), notes(8), popen(3S), system(3S),
The Notesfile Reference Manual

AUTHORS

Ray Essick (uiucdcs!essick, uiucdcs!notes)
Rob Kolstad (uiucdcs!kolstad)
Department of Computer Science
222 Digital Computer Laboratory
University of Illinois at Urbana-Champaign
1304 West Springfield Ave.
Urbana, IL 61801

NAME

nlist – get entries from name list

SYNOPSIS

```
#include <a.out.h>
nlist (file-name, nl)
char *file-name;
struct nlist nl[ ];
```

DESCRIPTION

Nlist examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See *a.out*(5) for a discussion of the symbol table structure.

This subroutine is useful for examining the system name list kept in the file /unix. In this way programs can obtain system addresses that are up to date.

SEE ALSO

a.out(5).

DIAGNOSTICS

All type entries are set to 0 if the file cannot be found or if it is not a valid namelist.

NAME

perror, sys_errlist, sys_nerr, errno – system error messages

SYNOPSIS

```
perror (s)
char *s;

int sys_nerr;
char *sys_errlist[ ];

int errno;
```

DESCRIPTION

Perror produces a short error message on the standard error, describing the last error encountered during a system call from a C program. First the argument string *s* is printed, then a colon, then the message and a new-line. To be of most use, the argument string should be the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur and cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new-line. *Sys_nerr* is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2).

NAME

plot – graphics interface subroutines

SYNOPSIS

openpl ()

erase ()

label (s)
char *s;

line (x1, y1, x2, y2)

circle (x, y, r)

arc (x, y, x0, y0, x1,

move (x, y)

cont (x, y)

point (x, y)

linemod (s)
char *s;

space (x0, y0, x1, y1)

closepl ()

DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. See *plot*(5) for a description of their effect. *Openpl* must be used before any of the others to open the device for writing. *Closepl* flushes the output.

String arguments to *label* and *linemod* are terminated by nulls and do not contain new-lines.

The library files listed below provide several flavors of these routines.

FILES

/usr/lib/libplot.a	produces output for *tplot*(1G) filters
/usr/lib/lib300.a	for DASI 300
/usr/lib/lib300s.a	for DASI 300s
/usr/lib/lib450.a	for DASI 450
/usr/lib/lib4014.a	for Tektronix 4014

SEE ALSO

graph(1G), tplot(1G), plot(5).

NAME

popen, pclose – initiate I/O to/from a process

SYNOPSIS

```
#include <stdio.h>

FILE *popen (command, type)
char *command, *type;

int pclose (stream)
FILE *stream;
```

DESCRIPTION

The arguments to *popen* are pointers to null-terminated strings containing, respectively, a shell command line and an I/O mode, either **r** for reading or **w** for writing. *Popen* creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer that can be used (as appropriate) to write to the standard input of the command or read from its standard output.

A stream opened by *popen* should be closed by *pclose*, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type **r** command may be used as an input filter, and a type **w** as an output filter.

SEE ALSO

pipe(2), wait(2), fclose(3S), fopen(3S), system(3S).

DIAGNOSTICS

Popen returns a null pointer if files or processes cannot be created, or if the shell cannot be accessed.

Pclose returns −1 if *stream* is not associated with a "*popen*ed" command.

BUGS

Only one stream opened by *popen* can be in use at once.

Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be forestalled by careful buffer flushing, e.g. with *fflush*; see *fclose*(3S).

NAME

printf, fprintf, sprintf – output formatters

SYNOPSIS

```
#include <stdio.h>

int printf (format [ , arg ] ...  )
char *format;

int fprintf (stream, format [ , arg ] ...  )
FILE *stream;
char *format;

int sprintf (s, format [ , arg ] ...  )
char *s, format;
```

DESCRIPTION

Printf places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places "output", followed by the null character (\0) in consecutive bytes starting at *s; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the \0 in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more *args*. The results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag (see below) has been given) to the field width;

A *precision* that gives the minimum number of digits to appear for the **d**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e** and **f** conversions, the maximum number of significant digits for the **g** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (.) followed by a decimal digit string: a null digit string is treated as zero.

An optional **l** specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*.

A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *args* specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

−	The result of the conversion will be left-justified within the field.
+	The result of a signed conversion will always begin with a sign (+ or −).
blank	If the first character of a signed conversion is not a sign, a blank will be prepended to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.
#	This flag specifies that the value is to be converted to an "alternate form." For **c**, **d**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** (**X**) conversion, a non-zero result will have **0x** (**0X**) prepended to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,o,u,x,X	The integer *arg* is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string (unless the conversion is **o**, **x**, or **X** *and* the **#** flag is present).
f	The float or double *arg* is converted to decimal notation in the style "[−]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are output; if the precision is explicitly 0, no decimal point appears.
e,E	The float or double *arg* is converted in the style "[−]d.ddde±dd", where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains exactly two digits.
g,G	The float or double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than −4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.
c	The character *arg* is printed.
s	The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (**\0**) is

encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed.

% Print a %; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putchar* had been called (see *putc*(3S)).

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %.2d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4•atan(1.0));
```

SEE ALSO

ecvt(3C), putc(3S), scanf(3S), stdio(3S).

NAME

putc, putchar, fputc, putw – put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc (c, stream)
char c;
FILE *stream;

putchar (c)

fputc (c, stream)
FILE *stream;

putw (w, stream)
int w;
FILE *stream;
```

DESCRIPTION

Putc appends the character c to the named output *stream*. It returns the character written.

Putchar(c) is defined as *putc*(c, *stdout*).

Fputc behaves like *putc*, but is a genuine function rather than a macro; it may therefore be used as an argument. *Fputc* runs more slowly than *putc*, but takes less space per invocation.

Putw appends the word (i.e., integer) *w* to the output *stream*. *Putw* neither assumes nor causes special alignment in the file.

The standard stream *stdout* is normally buffered if and only if the output does not refer to a terminal; this default may be changed by *setbuf*(3S). The standard stream *stderr* is by default unbuffered unconditionally, but use of *freopen*(3S) will cause it to become unbuffered; *setbuf*, again, will set the state to whatever is desired. When an output stream is unbuffered information appears on the destination file or terminal as soon as written; when it is buffered many characters are saved up and written as a block. See also *fflush*(3S).

SEE ALSO

ferror(3S), fopen(3S), fwrite(3S), getc(3S), printf(3S), puts(3S).

DIAGNOSTICS

These functions return the constant **EOF** upon error. Since this is a good integer, *ferror*(3S) should be used to detect *putw* errors.

BUGS

Because it is implemented as a macro, *putc* treats incorrectly a *stream* argument with side effects. In particular, putc(c, *f++); doesn't work sensibly.

NAME

putpwent – write password file entry

SYNOPSIS

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

DESCRIPTION

Putpwent is the inverse of *getpwent*(3C). Given a pointer to a *passwd* structure created by *getpwent* (or *getpwuid*(3C) or *getpwnam*(3C)), *putpwuid* writes a line on the stream *f* which matches the format of /etc/passwd.

DIAGNOSTICS

Putpwent returns non-zero if an error was detected during its operation, otherwise zero.

NAME

puts, fputs – put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

Puts copies the null-terminated string *s* to the standard output stream *stdout* and appends a new-line character.

Fputs copies the null-terminated string *s* to the named output *stream*.

Neither routine copies the terminating null character.

DIAGNOSTICS

Both routines return **EOF** on error.

SEE ALSO

ferror(3S), fopen(3S), fwrite(3S), gets(3S), printf(3S), putc(3S).

NOTES

Puts appends a new-line, *fputs* does not.

NAME

qsort – quicker sort

SYNOPSIS

qsort (base, nel, width, compar)
char *base;
int nel, width;
int (*compar)( );

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine. It is called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

SEE ALSO

sort(1), bsearch(3C), lsearch(3C), strcmp(3C).

NAME

rand, srand – random number generator

SYNOPSIS

```
srand (seed)
unsigned seed;

rand ( )
```

DESCRIPTION

Rand uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to $2^{15}-1$.

The generator is reinitialized by calling *srand* with 1 as argument. It can be set to a random starting point by calling *srand* with whatever you like as argument.

NAME

rand, srand – Fortran uniform random-number generator

SYNOPSIS

integer i, j

call srand(i)
j = rand( )

DESCRIPTION

Srand takes its integer argument as the seed of a random-number generator, the values of which are returned through successive invocations of *rand*.

SEE ALSO

rand(3C).

NAME

regex, regcmp – regular expression compile/execute

SYNOPSIS

```
char *regcmp(string1[,string2, ...],(char *)0);
char *string1, *string2, ...;

char *regex(re,subject[,ret0, ...]);
char *re, *subject, *ret0, ...;
```

DESCRIPTION

Regcmp compiles a regular expression and returns a pointer to the compiled form. *Malloc*(3C) is used to create space for the vector. It is the user's responsibility to free unneeded space so allocated. A zero return from *regcmp* indicates an incorrect argument. *Regcmp*(1) has been written to generally preclude the need for this routine at execution time.

Regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. *Regex* returns zero on failure or a pointer to the next unmatched character on success. A global character pointer *__loc1* points to where the match began. *Regcmp* and *regex* were mostly borrowed from the editor, *ed*(1) however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[ ]*.^	These symbols retain their current meaning.
$	Matches the end of the string, \n matches the new-line.
–	Within brackets the minus means *through*. For example, [a–z] is equivalent to [abcd...xyz]. The – can appear as itself only if used as the last or first character. For example, the character class expression [ ]–] matches the characters] and –.
+	A regular expression followed by + means *one or more times*. For example, [0–9]+ is equivalent to [0–9][0–9]*.
{m} {m,} {m,u}	Integer values enclosed in { } indicate the number of times the preceding regular expression is to be applied. *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (e.g., {m}), it indicates the exact number of times the regular expression is to be applied. {m,} is analogous to {m,infinity}. The plus (+) and star (*) operations are equivalent to {1,} and {0,} respectively.
(...)$*n*	The value of the enclosed regular expression is to be returned. The value will be stored in the *(n+1)*th argument following the subject argument. At present, at most ten enclosed regular expressions are allowed. *Regex* makes its assignments unconditionally.
(...)	Parentheses are used for grouping. An operator, e.g. *, +, { }, can work on a single character or a regular expression enclosed in parenthesis. For example, (a*(cb+)*)$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped to be used as themselves.

EXAMPLES

Example 1:

```
char *cursor, *newcursor, *ptr;
          ...
newcursor = regex((ptr=regcmp("^\n",(char *)0)),cursor);
free(ptr);
```

This example will match a leading new-line in the subject string pointed at by cursor.

Example 2:

```
char ret0[9];
char *newcursor, *name;
          ...
name = regcmp("([A-Za-z][A-za-z0-9_]{0,7})$0",(char *)0);
newcursor = regex(name,"123Testing321",ret0);
```

This example will match through the string "Testing3" and will return the address of the character after the last matched character (cursor+11). The string "Testing3" will be copied to the character array *ret0*.

Example 3:

```
#include "file.i"
char *string, *newcursor;
          ...
newcursor = regex(name,string);
```

This example applies a precompiled regular expression in **file.i** (see *regcmp*(1)) against *string*.

This routine is kept in /lib/libPW.a.

SEE ALSO

ed(1), regcmp(1), free(3C), malloc(3C).

BUGS

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required. The following user-supplied replacement for *malloc*(3C) re-uses the same vector saving time and space:

```
/* user's program */
          ...
malloc(n) {
static int rebuf[256];
          return &rebuf;
}
```

NAME

anint, dnint, nint, idnint – Fortran nearest integer functions

SYNOPSIS

```
integer i
real r1, r2
double precision dp1, dp2

r2 = anint(r1)
i = nint(r1)

dp2 = anint(dp1)
dp2 = dnint(dp1)

i = nint(dp1)
i = idnint(dp1)
```

DESCRIPTION

Anint returns the nearest whole real number to its real argument (i.e., int(a+0.5) if a ≥ 0, int(a–0.5) otherwise). *Dnint* does the same for its double-precision argment. *Nint* returns the nearest integer to its real argument. *Idnint* is the double-precision version. *Anint* is the generic form of *anint* and *dnint* , performing the same operation and returning the data type of its argument. *Nint* is also the generic form of *idnint*.

NAME

scanf, fscanf, sscanf – formatted input conversion

SYNOPSIS

#include <stdio.h>

scanf (format [, pointer] ... **)**
char *format;

fscanf (stream, format [, pointer] ... **)**
FILE *stream;
char *format;

sscanf (s, format [, pointer] ... **)**
char *s, *format;

DESCRIPTION

Scanf reads from the standard input stream *stdin*. *Fscanf* reads from the named input *stream*. *Sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs, or new-lines, which cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character **%**, an optional assignment suppressing character *****, an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *****. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

%	a single **%** is expected in the input at this point; no assignment is done.
d	a decimal integer is expected; the corresponding argument should be an integer pointer.
o	an octal integer is expected; the corresponding argument should be an integer pointer.
x	a hexadecimal integer is expected; the corresponding argument should be an integer pointer.
s	a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating **\0**, which will be added automatically. The input field is terminated by a space

```
        56789 0123 56a72
```

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* (see *getc*(3S)) will return **a**.

SEE ALSO

atof(3C), getc(3S), printf(3S).

NOTE

Trailing white space (including a new-line) is left unread unless matched in the control string.

DIAGNOSTICS

These functions return **EOF** on end of input and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

NAME

setbuf – assign buffering to a stream

SYNOPSIS

#include <stdio.h>

setbuf (stream, buf)
FILE *stream;
char *buf;

DESCRIPTION

Setbuf is used after a stream has been opened but before it is read or written. It causes the character array *buf* to be used instead of an automatically allocated buffer. If *buf* is the constant pointer **NULL**, input/output will be completely unbuffered.

A manifest constant **BUFSIZ** tells how big an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from *malloc*(3C) upon the first *getc* or *putc*(3S) on the file, except that output streams directed to terminals, and the standard error stream *stderr* are normally not buffered.

A common source of error is allocation of buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

SEE ALSO

fopen(3S), getc(3S), malloc(3C), putc(3S).

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

#include <setjmp.h>

int setjmp (env)
jmp_buf env;

longjmp (env, val)
jmp_buf env;

DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* for later use by *longjmp*. It returns value 0.

Longjmp restores the environment saved by the last call of *setjmp*. It then returns in such a way that execution continues as if the call of *setjmp* had just returned the value *val* to the corresponding call to *setjmp*, which must not itself have returned in the interim. *Longjmp* cannot return the value 0. If *longjmp* is invoked with a second argument of 0, it will return 1. All accessible data have values as of the time *longjmp* was called.

SEE ALSO

signal(2).

NAME

sign, isign, dsign – Fortran transfer-of-sign intrinsic function

SYNOPSIS

```
integer i, j, k
real r1, r2, r3
double precision dp1, dp2, dp3

k = isign(i, j)
k = sign(i, j)

r3 = sign(r1, r2)

dp3 = dsign(dp1, dp2)
dp3 = sign(dp1, dp2)
```

DESCRIPTION

Isign returns the magnitude of its first argument with the sign of its second argument. *Sign* and *dsign* are its real and double-precision counterparts, respectively. The generic version is *sign* and will devolve to the appropriate type depending on its arguments.

NAME

signal – specify Fortran action on receipt of a system signal

SYNOPSIS

integer i
external integer intfnc

call signal(i, intfnc)

DESCRIPTION

Signal allows a process to specify a function to be invoked upon receipt of a specific signal. The first argument specifies which fault or exception, the second argument the function to be invoked.

SEE ALSO

kill(2), signal(2).

NAME

sin, dsin, csin – Fortran sine intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2 = sin(r1)

dp2 = dsin(dp1)
dp2 = sin(dp1)

cx2 = csin(cx1)
cx2 = sin(cx1)
```

DESCRIPTION

Sin returns the real sine of its real argument. *Dsin* returns the double-precision sine of its double-precision argument. *Csin* returns the complex sine of its complex arguemnt. The generic *sin* function becomes *dsin* or *csin* as required by argument type.

SEE ALSO

trig(3M).

NAME

sinh, dsinh – Fortran hyperbolic sine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = sinh(r1)

dp2 = dsinh(dp1)
dp2 = sinh(dp1)

DESCRIPTION

Sinh returns the real hyperbolic sine of its real argument. *Dsinh* returns the double-precision hyperbolic sine of its double-precision argument. The generic form *sinh* may be used to return a double-precision value given a double-precision argument.

SEE ALSO

sinh(3M).

NAME

sinh, cosh, tanh – hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;
```

DESCRIPTION

Sinh, *cosh* and *tanh* return respectively the hyberbolic sine, cosine and tangent of their argument.

DIAGNOSTICS

Sinh and *cosh* return **HUGE** when the correct value would overflow, and set *errno* to ERANGE.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

sleep – suspend execution for interval

SYNOPSIS

unsigned sleep (seconds)
unsigned seconds;

DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) Because scheduled wakeups occur at fixed 1-second intervals, and (2) because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*; if the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred, and the caller's alarm catch routine is executed just before the *sleep* routine returns, but if the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

SEE ALSO

alarm(2), pause(2), signal(2).

NAME

sqrt, dsqrt, csqrt – Fortran square root intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2 = sqrt(r1)

dp2 = dsqrt(dp1)
dp2 = sqrt(dp1)

cx2 = csqrt(cx1)
cx2 = sqrt(cx1)

DESCRIPTION

Sqrt returns the real square root of its real argument. *Dsqrt* returns the double-precision square root of its double-precision arguement. *Csqrt* returns the complex square root of its complex argument. *Sqrt*, the generic form, will become *dsqrt* or *csqrt* as required by its argument type.

SEE ALSO

exp(3M).

NAME

ssignal, gsignal – software signals

SYNOPSIS

```
#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;
```

DESCRIPTION

Ssignal and *gsignal* implement a software facility similar to *signal*(2). This facility is used by the Standard C Library to enable the user to indicate the disposition of error conditions, and is also made available to the user for his own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 15. An *action* for a software signal is *established* by a call to *ssignal*, and a software signal is *raised* by a call to *gsignal*. Raising a software signal causes the action established for that signal to be *taken*.

The first argument to *ssignal* is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user defined) *action function* or one of the manifest constants **SIG_DFL** (default) or **SIG_IGN** (ignore). *Ssignal* returns the action previously established for that signal type; if no action has been established or the signal number is illegal, *ssignal* returns **SIG_DFL**.

Gsignal raises the signal identified by its argument, *sig*:

> If an action function has been established for *sig*, then that action is reset to **SIG_DFL** and the action function is entered with argument *sig*. *Gsignal* returns the value returned to it by the action function.
>
> If the action for *sig* is **SIG_IGN**, *gsignal* returns the value 1 and takes no other action.
>
> If the action for *sig* is **SIG_DFL**, *gsignal* returns the value 0 and takes no other action.
>
> If *sig* has an illegal value or no action was ever specified for *sig*, *gsignal* returns the value 0 and takes no other action.

NOTES

There are some additional signals with numbers outside the range 1 through 15 which are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 15 are legal, although their use may interfere with the operation of the Standard C Library.

[illegible]

NAME

stdio – standard buffered input/output package

SYNOPSIS

```
#include <stdio.h>
FILE *stdin, *stdout, *stderr;
```

DESCRIPTION

The functions described in the entries of sub-class 3S of this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar*, *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type **FILE**. *Fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are 3 open streams with constant pointers declared in the "include" file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file.

A constant "pointer" **NULL** ((char *)0) designates the null stream.

An integer constant **EOF** (–1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

```
#include <stdio.h>
```

The functions and constants mentioned in the entries of sub-class 3S of this manual are declared in that "include" file and need no further declaration. The constants and the following "functions" are implemented as macros (redeclaration of these names is perilous): *getc*, *getchar*, *putc*, *putchar*, *feof*, *ferror*, and *fileno*.

SEE ALSO

open(2), close(2), read(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), system(3S), tmpnam(3S).

DIAGNOSTICS

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

NAME

stdout – Write unformatted data to stdout

SYNOPSIS

character*N string

call stdout(string)

DESCRIPTION

Stdout writes the entire string to the stdout-unit (no. 6 in f77-programs)

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok – string operations

SYNOPSIS

```
char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s, c;

char *strrchr (s, c)
char *s, c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;
```

DESCRIPTION

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

Strcat appends a copy of string *s2* to the end of string *s1*. *Strncat* copies at most *n* characters. Both return a pointer to the null-terminated result.

Strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2*. *Strncmp* makes the same comparison but looks at at most *n* characters.

Strcpy copies string *s2* to *s1*, stopping after the null character has been moved. *Strncpy* copies exactly *n* characters, truncating or null-padding *s2*; the target may not be null-terminated if the length of *s2* is *n* or more. Both return *s1*.

Strlen returns the number of non-null characters in *s*.

Strchr (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or **NULL** if *c* does not occur in the string. The null character terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or **NULL** if no character from *s2* exists in *s1*.

Strspn (*strcspn*) returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*.

Strtok considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a **NULL** character into *s1* immediately following the returned token. Subsequent calls with zero for the first argument, will work through the string *s1* in this way until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a **NULL** is returned.

BUGS

Strcmp uses native character comparison, which is signed on PDP-11s or CADMUS-9000s, unsigned on other machines.

All string movement is performed character by character starting at the left. Thus overlapping moves toward the left will work as expected, but overlapping moves to the right may yield surprises.

NAME

swab – swap bytes

SYNOPSIS

swab (from, to, nbytes)
char *from, *to;
int nbytes;

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP-11s and other machines. *Nbytes* should be even.

NAME
: system – issue a shell command from Fortran

SYNOPSIS

character*N c

call system(c)

DESCRIPTION

System causes its character argument to be given to *sh*(1) as input, as if the string had been typed at a terminal. The current process waits until the shell has completed.

SEE ALSO

sh(1), exec(2), system(3S).

NAME

system – issue a shell command

SYNOPSIS

```
#include <stdio.h>

int system (string)
char *string;
```

DESCRIPTION

System causes the *string* to be given to *sh*(1) as input as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

SEE ALSO

sh(1), exec(2).

DIAGNOSTICS

System stops if it can't execute *sh*(1).

NAME
tan, dtan – Fortran tangent intrinsic function

SYNOPSIS
```
real r1, r2
double precision dp1, dp2

r2 = tan(r1)

dp2 = dtan(dp1)
dp2 = tan(dp1)
```

DESCRIPTION
Tan returns the real tangent of its real argument. *Dtan* returns the double-precision tangent of its double-precision argument. The generic *tan* function becomes *dtan* as required with a double-precision argument.

SEE ALSO
trig(3M).

NAME

tanh, dtanh – Fortran hyperbolic tangent intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2

r2 = tanh(r1)

dp2 = dtanh(dp1)
dp2 = tanh(dp1)
```

DESCRIPTION

Tanh returns the real hyperbolic tangent of its real argument. *Dtanh* returns the double-precision hyperbolic tangent of its double precision argument. The generic form *tanh* may be used to return a double-precision value given a double-precision argument.

SEE ALSO

sinh(3M).

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs – terminal independent operation routines

SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, **area;

char *
tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

DESCRIPTION

These functions extract and use capabilities from the terminal capability data base *termcap*(5). These are low level routines; see *curses*(3) for a higher level package.

Tgetent extracts the entry for terminal *name* into the buffer at *bp*. *Bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to *tgetnum*, *tgetflag*, and *tgetstr*. *Tgetent* returns −1 if it cannot open the *termcap* file, 0 if the terminal name given does not have an entry, and 1 if all goes well. It will look in the environment for a TERMCAP variable. If found, and the value does not begin with a slash, and the terminal type **name** is the same as the environment string TERM, the TERMCAP string is used instead of reading the termcap file. If it does begin with a slash, the string is used as a path name rather than */etc/termcap*. This can speed up entry into programs that call *tgetent*, as well as to help debug new terminal descriptions or to make one for your terminal if you can't write the file */etc/termcap*.

Tgetnum gets the numeric value of capability *id*, returning −1 if is not given for the terminal. *Tgetflag* returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. *Tgetstr* gets the string value of capability *id*, placing it in the buffer at *area*, advancing the *area* pointer. It decodes the abbreviations for this field described in *termcap*(5), except for cursor addressing and padding information.

Tgoto returns a cursor addressing string decoded from **cm** to go to column *destcol* in line *destline*. It uses the external variables **UP** (from the **up** capability) and BC (if **bc** is given rather than **bs**) if necessary to avoid placing **\n**, ^D or ^@ in the returned string. (Programs which call tgoto should be sure to turn off the XTABS bit(s), since tgoto may now output a tab. Note that programs using termcap should in general turn off XTABS anyway since some terminals use control I for other functions, such as nondestructive space.) If a **%** sequence is given which is not understood, then *tgoto* returns OOPS.

Tputs decodes the leading padding information of the string *cp; affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine which is called with each character in turn. The external variable *ospeed* should contain the output speed of the terminal as encoded by *ioctl (2)*. The external variable PC should contain a pad character to be used (from the **pc** capability) if a null (^@) is inappropriate.

FILES

/usr/lib/libtermcap.a	–ltermcap library
/etc/termcap	data base

SEE ALSO

ex(1), curses(3), termcap(5)

AUTHOR

William Joy

BUGS

NAME

tmpfile – create a temporary file

SYNOPSIS

#include <stdio.h>

FILE *tmpfile ()

DESCRIPTION

Tmpfile creates a temporary file and returns a corresponding **FILE** pointer. Arrangements are made so that the file will automatically be deleted when the process using it terminates. The file is opened for update.

SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3C), tmpnam(3S).

NAME

tmpnam – create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam (s)
char *s;
```

DESCRIPTION

Tmpnam generates a file name that can safely be used for a temporary file. If (int)*s* is zero, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If (int)*s* is nonzero, *s* is assumed to be the address of an array of at least **L_tmpnam** bytes; *tmpnam* places its result in that array and returns *s* as its value.

Tmpnam generates a different file name each time it is called.

Files created using *tmpnam* and either *fopen* or *creat* are only temporary in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink* (2) to remove the file when its use is ended.

SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3C).

BUGS

If called more than 17,576 times in a single process, *tmpnam* will start recycling previously used names.
Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using *tmpnam* or *mktemp*, and the file names are chosen so as to render duplication by other means unlikely.

NAME

sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin (x)
double x;

double cos (x)
double x;

double tan (x)
double x;

double asin (x)
double x;

double acos (x)
double x;

double atan (x)
double x;

double atan2 (y, x)
double x, y;
```

DESCRIPTION

Sin, *cos* and *tan* return respectively the sine, cosine and tangent of their argument, which is in radians.

Asin returns the arcsine of x, in the range $-\pi/2$ to $\pi/2$.

Acos returns the arccosine of x, in the range 0 to π.

Atan returns the arctangent of x, in the range $-\pi/2$ to $\pi/2$.

Atan2 returns the arctangent of y/x, in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the return value.

DIAGNOSTICS

Sin, *cos* and *tan* lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return 0 when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments, a PLOSS error is generated but no message is printed. In both cases, *errno* is set to **ERANGE**.

Tan returns **HUGE** for an argument which is near an odd multiple of $\pi/2$ when the correct value would overflow, and sets *errno* to **ERANGE**.

Arguments of magnitude greater than 1.0 cause *asin* and *acos* to return 0 and to set *errno* to **EDOM**. In addition, a message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

tsearch, tdelete, twalk – manage binary search trees

SYNOPSIS

#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );

DESCRIPTION

Tsearch is a binary tree search routine generalized from Knuth (6.2.2) Algorithm T. It returns a pointer into a tree indicating where a datum may be found. If the datum does not occur, it is added at an appropriate point in the tree. *Key* points to the datum to be sought in the tree. *Rootp* points to a variable that points to the root of the tree. A NULL pointer value for the variable denotes an empty tree; in this case, the variable will be set to point to the datum at the root of the new tree. *Compar* is the name of the comparison function. It is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according as the first argument is to be considered less than, equal to, or greater than the second.

Tdelete deletes a node from a binary search tree. It is generalized from Knuth (6.2.2) algorithm D. The arguments are the same as for *tsearch*. The variable pointed to by *rootp* will be changed if the deleted node was the root of the tree. *Tdelete* returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

Twalk traverses a binary search tree. *Root* is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *Action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum { preorder, postorder, endorder, leaf } VISIT;* (defined in the <search.h> header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

NOTES

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.
Warning: the *root* argument to *twalk* is one level of indirection less than the *rootp* arguments to *tsearch* and *tdelete*.

DIAGNOSTICS

A NULL pointer is returned by *tsearch* if there is not enough space available to create a new node.

A NULL pointer is returned by *tsearch* and *tdelete* if *rootp* is NULL on entry.

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

BUGS

Awful things can happen if the calling function alters the pointer to the root.

NAME

ttyname, isatty – find name of a terminal

SYNOPSIS

char *ttyname (fildes)

int isatty (fildes)

DESCRIPTION

Ttyname returns a pointer to the null-terminated path name of the terminal device associated with file descriptor *fildes*.

Isatty returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

FILES

/dev/*

DIAGNOSTICS

Ttyname returns a null pointer (0) if *fildes* does not describe a terminal device in directory **/dev**.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

ttyslot – find the slot in the utmp file of the current user

SYNOPSIS

int ttyslot ( )

DESCRIPTION

Ttyslot returns the index of the current user's entry in the **/etc/utmp** file. This is accomplished by actually scanning the file **/etc/inittab** for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1 or 2).

FILES

/etc/inittab
/etc/utmp

SEE ALSO

getut(3C), ttyname(3C).

DIAGNOSTICS

A value of 0 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

NAME

ungetc – push character back into input stream

SYNOPSIS

#include <stdio.h>

int ungetc (c, stream)
char c;
FILE *stream;

DESCRIPTION

Ungetc pushes the character **c** back on an input stream. That character will be returned by the next *getc* call on that stream. *Ungetc* returns **c**.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push **EOF** are rejected.

Fseek(3S) erases all memory of pushed back characters.

SEE ALSO

fseek(3S), getc(3S), setbuf(3S).

DIAGNOSTICS

Ungetc returns **EOF** if it can't push a character back.

NAME

intro – introduction to special files

DESCRIPTION

This section describes various special files that refer to specific hardware peripherals and UNIX System device drivers. The names of the entries are generally derived from names for the hardware, as opposed to the names of the special files themselves. Characteristics of both the hardware device and the corresponding UNIX System device driver are discussed where applicable.

BUGS

While the names of the entries *generally* refer to vendor hardware names, in certain cases these names are seemingly arbitrary for various historical reasons.

NAME

bbp – Basic Block Port Interface

SYNOPSIS

#include <sys/port.h>

DESCRIPTION

The Basic Block Port Interface is a simple network interface, originally developed for the Cambridge Ring, and now adapted to Ethernet. The bbp provides a set of so-called ports, through which processes on different machines (or on the same) can talk to each other. The port is characterized by the structure portinfo in *<sys/port.h>* :

```
struct portinfo {
        short           pi_type;        /* port type,unused for Ethernet */
        unsigned int    pi_inport;      /* bb_port number by which this port
                                         * is addressed by other stations
                                         */
        short           pi_station;     /* destination ring station */
        unsigned int    pi_outport;     /* destination bb_port number */
        unsigned int    pi_accept;      /* acceptable source station number */
        etheradr        pi_ethadr;      /* destination ethernet address */
};
```

The field *pi_type* is unused for Ethernet. For the Cambridge ring this field specifies if the data transfers are protected by parity checks or not. The field *pi_inport* specifies the input port number, by which this port is addressed by other stations. Their output port number, *pi_outport*, must be equal to pi_inport, if they want to talk to this port. The field *pi_station* is a two-byte station number.

A station number is a unique identification of each machine attached to the net. This station number is more manageable than the six byte Ethernet address contained in the field *pi_ethadr*. The ethernet address is mainly used for the hardware address recognition, whereas the station number is used by upper level software. Both the station number and the ethernet address are fixed at system generation time. They are, like the ascii system name, a unique name of the system. Their values can be found in the file */usr/sys/name.c* and can be gotten by the system call *uname*(2).

The fields *pi_station*, and *pi_ethadr* together specify the destination machine; the field *pi_outport* is the port number on the destination machine, to which this port wants to talk. The field *pi_accept* specifies the machines from which this port is willing to receive. The values NOONE and ANYONE mean: accept packets from noone or anyone. Any other number means: accept packets only from the station with this number.

It is not necessary for a process to specify his own station number and ethernet address, as the system knows them already and they cannot change.

The portinfo structure is set by an ioctl system call with command BBPSET, and read with command BBPGET.

EXAMPLE

Machine alpha has the station number 3 and the ethernet address 333333333333. Machine beta has the station number 5 and the ethernet address 555555555555. A process on alpha wishes to receive on port number 372. Another process on beta receives on port number 373; The process on alpha specifies

```
struct portinfo alphaport = { 0, 372, 5, 373, 5, {0x5555,0x5555,0x5555}};
        ioctl(fd,BBPSET,&alphaport);
```

whereas the process on beta specifies

```
struct portinfo betaport = { 0, 373, 3, 372, 3, {0x3333,0x3333,0x3333}};
        ioctl(fd,BBPSET,&betaport);
```

Both processes can now talk to each other by normal read and write system calls.

A port can be obtained by successively opening the files /dev/bbp0, /dev/bbp1 etc. If the open returns with errno ENXIO, the port does not exist, if errno is EACCES, the port is already opened. After the open the port must be configured with the command BBPSET. If the ioctl returns with errno EACCES, a port with the same *pi_inport* is already open. Just to get an unused port number, the value DYNAMIC can be given for *pi_inport*. The actual port number can then be gotten with the ioctl-command BBPGET.

EXAMPLE

```
struct portinfo aport = { 0, DYNAMIC, 5, 123, ANYONE, {0x5555,0x5555,0x5555}};
        ioctl(fd,BBPSET,&aport);
        ioctl(fd,BBPGET,&aport);  /* pi_inport contains a free port number */
```

Data is transferred with the normal read and write system calls. However, there is a limitation on the number of bytes that can be transferred with one write. On the Ethernet, the number 1024 is safe. The data of each write system call is sent as a packet over the net. The count of the read system call must be larger or equal than the size of the packet, otherwise the read returns with error EIO. read returns the size of the received packet. If the count for read or write is illegal, error EINVAL is returned.

At any time after BBPSET, the ioctl command BBPENQ will return the following structure, defined in *<sys/port.h>*:

```
struct porteng {
        short           pn_sender;      /* station number
                                           of sender of received block */
        short           pn_sendport;    /* port number
                                           of sender of received block */
        char            pn_xrslt;       /* last block transmission result */
        char            pn_blkavail;    /* a block is available to be read */
        etheradr        pn_sendadr;     /* ethernet address of sender */
};
```

The fields *pn_sender*, *pn_sendport*, and *pn_sendadr* specify the station number, port number, and ethernet address of the sender of a received packet. The field *pn_xrslt* contains the result of the last write. This is normally equal to BB_ACCEPTED on the ethernet, and equal to BB_ERROR only if excessive jams occurred on the net. The field *pn_blkavail* is unequal 0, if a packet has been received, but not yet read.

WARNING

The bbp contains no flow control. Incoming packets are simply discarded if they are not read fast enough. Protocols are entirely the responsibility of upper levels.

SEE ALSO

sbp(4)

FILES

```
/dev/bbp*
```

NAME

bip – CADMUS Bitmap Display

DESCRIPTION

The bip can be accessed in two different ways. First, it can be used as a normal teletype, with only one font set and no graphics, but with a high number of lines and columns. The interface to this mode consists of the normal open/close/read/write/ioctl system calls. The second mode views the bip as a piece of memory of size 256k. Reading and writing of this memory causes dots on the screen to turn black or white, or it transfers parameters to a program which executes locally on the bip.

For the first mode, the discussion of typewriter I/O given in *termio*(4) applies. An additional ioctl call TCBIPADR has been provided in <*sys/termio.h*> that returns the starting address of the 256k memory of the bip.

The bip driver talks to code installed on the ROM of the bip, that partly emulates a VT52/VT100. The emulation understands the control sequences described in <*bip/vt100.h*>. An entry for bip is available in */etc/termcap*. If the bip is equipped with a keyboard, it can be used as any other terminal by making an entry for it in */etc/inittab*.

EXAMPLE

```
int bip; char *bipaddr;

bip = open("/dev/ttyb0", 2);
write(bip, "hallo0, 6);
ioctl(bip, TCBIPADR, &bipaddr);
bipaddr[xx] = 0xCC;
```

A possible entry in /etc/inittab:

```
tb0:2:respawn: /etc/getty ttyb0 9600 bip
```

FILES

/usr/include/bip/*
/dev/ttyb0, /dev/ttyb1, ...

SEE ALSO

bip(3), termcap(5), inittab(5)

NAME

configuration information – table of interrupt vector and device addresses

SYNOPSIS

cat /usr/sys/confinfo

DESCRIPTION

Confinfo is a table of the interrupt vectors and device addresses used in **MUNIX**. Addresses are listed both in octal and hex notation.

Please note that the octal values of interrupt vector addresses have to be used to switch DEC or DEC-compatible controller boards. The MC68000 processor assumes them to be interrupt vector numbers and computes the memory address by multiplying with 4 (resulting in the hex values).

Configuration Information

Device	Interrupt Vec.		Device Address		max. Units/Lines
	octal	hex	octal	hex	
console	60	C0	777560	FFFF70	1
ether	100	100	764330	FFE8D8 rx	1
	104	110	764332	FFE8DA tx	
	110	120			
lbp	120	140	770000	FFF000	1
bip	134	170		0	1
kedqs			770400	FFF100	4 Lines
port 0	140	180	770440	FFF120-14F	
port 1	144	190	770520	FFF150-17F	
port 2	150	1A0	770600	FFF180-1AF	
port 3	154	1B0	770660	FFF1B0-1FF	
rl	160	1C0	774400	FFF900	4 Drives
hl	164	1D0	774420	FFF910	4 Drives
vp	174	1F0 plot	777400	FFFF00	1
	204	210 print			
lp	200	200	777514	FFFF4C	1
hk	210	220	777440	FFFF20	8 Drives
rk	220	240	777400	FFFF00	4 Drives
tm	224	250	772520	FFF550	8 Drives
ot/ox	230	260	775600	FFFB80 csr	4 Drives
			775640	FFFBA0 data	
st	240	280	777600	FFFF80 csr	1 Drive
alter.			777640	FFFFA0 data	

Configuration Information					
Device	Interrupt Vec.		Device Address		max. Units/Lines
	octal	hex	octal	hex	
rm	254	2B0	776700	FFFDC0	8 Drives
hp	254	2B0	776700	FFFDC0	2 Drives
rx2	264	2D0	777170	FFFE78	2 Drives
hx2	270	2E0	777150	FFFE68	2 Drives
st	270	2E0	777600	FFFF80 csr	1 Drive
			777640	FFFFA0 data	
tty					8 Lines
1	300	300	776500	FFFD40	
2	310	320	776510	FFFD48	
3	320	340	776520	FFFD50	
4	330	360	776530	FFFD58	
5	340	380	776540	FFFD60	
6	350	3A0	776550	FFFD68	
7	360	3C0	776560	FFFD70	
8	370	3E0	776570	FFFD78	
slu					7 Lines and Console (port 0)
0	304	310	776040	FFFC20	
1	300	300	776000	FFFC00	
2	314	330	776140	FFFC60	
3	310	320	776100	FFFC40	
4	324	350	776240	FFFCA0	
5	320	340	776200	FFFC80	
6	334	370	776340	FFFCE0	
7	330	360	776300	FFFCC0	
dz(v)					32(16) Lines
1st	330	360	760100	FFE040	
2nd	340	380	760110	FFE048	
3rd	350	3A0	760120	FFE050	
4th	360	3C0	760130	FFE058	
dh	340	380	760020	FFE010	16 Lines
td	370	3E0	777600	FFFF80 csr	2 Drives
			777640	FFFFA0 data	

FILES

/usr/sys/confinfo

NAME

dz, dh – DZ-11, DH-11 asynchronous multiplexers

DESCRIPTION

Each line attached to a DH-11 or DZ-11 communications multiplexer behaves as described in *termio*(4). Input and output for each line may be set to run at any of 16 speeds; see *termio*(4) for the encoding. (For DZ-11 lines, output speed is always the same as input speed. The *200* speed and the two externally clocked speeds (*exta*, *extb*) are missing on the DZ-11.)

FILES

/dev/tty??

SEE ALSO

termio(4).

NAME

hk – RK611/RK06, RK07 moving-head disk

DESCRIPTION

The octal representation of the minor device number is encoded *dp*, where *d* is a physical drive number, and *p* is a logical unit (subsection) within a physical unit. The origins and sizes of the logical units on each drive, counted in cylinders of 66 512-byte blocks and tracks of 22 512-byte blocks, are:

logical unit	starting cylinder	length (cyl.+tracks)	size (in blocks)	
0	0	146+0	9636	(root on RK06/07)
1	146	135+0	8910	(swap on RK06/07)
2	281	127+2	8426	(rest of RK06)
3	281	531+2	35090	(rest of RK07)
4	0	0+0	0	(spare)
5	0	0+0	0	(spare)
6	0	408+2	26972	(all of RK06)
7	0	812+2	53636	(all of RK07)

Systems distributed for these devices use disk 0 for the root, disk 1 for swapping, and disk 6 (RK06) or disk 7 (RK07) for a mounted user file system.

The embedded bad sector replacement mechanism assumes a bad sector file on each disk volume. Before using any disk volums please check them for bad sectors with the CADMUS disk check program. For the root filesystem and the swap area we suggest to use only *bad-sector-free* disks. Otherwise the system also works well but it will slow down. If you have bad sectors in the root filesystem area you cannot boot **MUNIX** directly from the *Minitor*. The Minitor doesn't handle bad sectors. In this case you have to boot from any other device (i.e. floppy, streamer ...). Only the standalone driver and the MUNIX-driver can handle bad sectors.

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records.

A 'raw' interface provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.' In raw I/O the buffer must begin on a word boundary.

Under **MUNIX** disk volumes may be changed without rebooting the system. In standalone mode however you have to restart the standalone program after changing a disk volume.

FILES

/dev/hk?, /dev/rhk?

SEE ALSO

format(8), check(8), iopage(7)
Bad Sector Handling (Vol. 2c)

DIAGNOSTICS

Sector numbers in error diagnostics are absolute. The sector numbers range from 0 to 27126 for an RK06 and from 0 to 53790 for an RK07. The last 154 sectors are not visible to the user. They are reserved for bad sector handling. The message *Cannot read bad sector file* using a disk volume is fatal. Do not use this disk any longer. The message *No replace sector available* indicates a new bad sector. Use the printed sector number to update the bad sector information by the disk check program.

NAME

hp – RP04/05/06, RM02/03 moving-head disk

DESCRIPTION

The octal representation of the minor device number is encoded *dp*, where *d* is a physical drive number, and *p* is a logical unit (subsection) within a physical unit. The origins and sizes of the logical units on each drive, counted in cylinders of 418 512-byte blocks, for the RP04/05/06 are:

log. unit	start cyl.	length	size (in blocks)	
0	0	23	9614	(root on RP04/05/06)
1	23	21	8778	(swap on RP04/05/06)
2	44	21	8778	(/sys on RP04/05/06)
3	65	345	144210	(rest of RP04/05)
4	65	749	313082	(rest of RP06)
5	411	403	168454	(/usr on RP06)
6	0	410	171380	(all of RP04/05)
7	0	814	340252	(all of RP06)

The logical units for the RM02/03 are:

log. unit	start cyl.	length	size (in blocks)	
0	0	60	9600	(root on RM02/03)
1	60	55	8800	(swap on RM02/03)
2	115	50	8000	(/sys on RM02/03)
3	165	657	105120	(rest of RM02/03)
4	0	0	0	(spare)
5	0	0	0	(spare)
6	0	0	0	(spare)
7	0	822	131520	(all of RM02/03)

Systems distributed for these devices use disk 0 for the root, disk 1 for swapping, and disk 3 (RP04/05, RM02/03) or disk 4 (RP06) for a mounted user file system.

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records.

A 'raw' interface provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.' In raw I/O the buffer must begin on a word boundary.

FILES

/dev/rp?, /dev/rrp?
/dev/rm?, /dev/rrm?

SEE ALSO

format (8)

BUGS

In raw I/O *read* and *write*(2) truncate file offsets to 512-byte block boundaries, and *write* scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, *read*, *write* and *lseek*(2) should always deal in 512-byte multiples.

NAME

kl – KL-11 or DL-11 asynchronous interface

DESCRIPTION

The discussion of typewriter I/O given in *termio*(4) applies to these devices.

KL stands normally for a DLV-11 J, DL for a DLV-11 E. The DL11-E supports modem control, the KL not. For the DLV-11 J, the console is attached to port 3. Port 0 to 2 correspond to /dev/tty1 to /dev/tty3. Make sure that the jumpers which cause a break to issue HALT or INIT on the Q-Bus are removed.

Attempts to change the speed are ignored.

FILES

/dev/console /dev/tty?

NAME

lbp – LBP-10 Laser Beam Printer Interface

SYNOPSIS

#include <sys/lbp.h>

DESCRIPTION

The Laser Printer is controlled by ioctl system calls. Parameter LBPPUT sets printer parameters, LBPGET asks for the current printer parameters, and LBPWRITE starts the actual printing. The settable parameters determine the rectangle on the paper which is to be printed, and whether the printer should use half resolution. For the purpose of this discussion we assume that the paper is divided into lines and columns, where the lines and columns are 0.1 mm apart. The lines and columns cut the paper into tiny 0.1 mm squares, called pixels. Each pixel on the paper can be black or white. We print by setting bits in memory to 0 or 1 for white or black. Lines on the paper correspond to sequential words in memory. Each line is a multiple of 16 pixels long. The upper left pixel is the most significant bit in the first word to be printed. Normally we do not want to print over the whole paper. The controller lets us specify a margin on the left and upper edge of the paper, and the number of lines and columns that make up the print area.

LBPPUT and LBPGET use the following structure, defined in *<sys/lbp.h>*:

```
struct lbp {
        short nla, nl, npa, np;
        short halfres;
};
```

The parameter nla is the number of print lines on top of the page to be left blank, nl is the number of lines to be printed, npa is the number of columns that constitute the left margin, and np is the length of the lines (number of columns). npa and np count multiples of 16, that is, npa == 1 means 1.6 mm left margin. If halfres is set different from 0, the resolution is changed from 0.1 mm to 0.2 mm.

The following code sets the parameters and asks for them:

```
        #include <sys/lbp.h>
        int l;
        struct lbp lbp = { ...,...,...,..., 0};
        int l;

        l = open("/dev/lbp",1);
        ioctl(l, LBPPUT, &lbp);   /* set params */
        ioctl(l, LBPGET, &lbp);   /* get params */
```

To print a page, a (very large) amount of memory has to be filled with pixels. The address and length of this area are given to an ioctl call with parameter LBPWRITE. LBPWRITE uses the following structure, again defined in *<sys/lbp.h>*:

```
struct lbpwrite {
        short *adr;
        long cnt;
};
```

The address must be even, the count is the count in words, not bytes! The call is like those above.

Diagnostics

The system call *open*(2) returns -1 and errno ENXIO when the printer is already opened, -1 and errno EIO when the printer is not ready (e.g. power off). When printing, the state of the printer is checked before and after printing. Before printing, if the printer is not ready, the driver will wait until the error condition is removed. If there is an error after printing, one retry will be made. Messages describing the state of the printer are sent to the system console.

NAME

lp − parallel line printer

DESCRIPTION

Lp provides the interface to any standard parallel line printer with Centronics interface. When it is opened or closed, a suitable number of page ejects is generated. Bytes written are printed.

In the default mode the driver correctly interprets carriage returns, backspaces, tabs, and form-feeds. A new-line that extends over the end of a page is turned into a form-feed. The default line length is 132 characters, indent is 4 characters and lines per page is 66. Lines longer than the line length minus the indent (i.e. 128 characters, using the above defaults) are truncated.

The command *lpctrl* can be used to change the driver mode, line length, lines per page and indent.

In *capital mode* lower case letters are turned into upper case and the characters { } ` | ~ are escaped by other symbols. In *transparent mode* all bytes written are printed without regarding line length, indent and page length.

There is an *ioctl(2)* call applying to the parallel line printer. It uses the following structure, defined in *<sys/lpcmd.h>:*

```
struct lpct {
        char aind;
        char aflg;
        int alin;
        int acol;
};
```

The *aind*, *alin*, *acol* and *aflg* fields describe the indent, page length, line length and driver mode. Symbolic values for driver modes are defined in *<sys/lpcmd.h>:*

CAP	020	capital mode
LPTRANS	0200	transparent mode

The *ioctl* call has the form:

```
#include <sys/lpcmd.h>

ioctl (fildes, command, arg)
struct lpct *arg;
```

The applicable commands are:

LPGET
: Fetch the parameters associated with the line printer, and store in the pointed-to structure.

LPSET Set the parameters according to the pointed-to structure.

FILES

/dev/lp
/usr/include/sys/lpcmd.h

SEE ALSO

lpr(1), lpctrl(1), ioctl(2).

NAME

mem, kmem – core memory

DESCRIPTION

Mem is a special file that is an image of the physical memory of the computer (excluding the I/O-page). *Kmem* is an imake of the logical memory including the I/O- page. Both files may be used, for example, to examine, and even to patch the system.

Addresses in *mem* are interpreted as memory addresses. References to non-existent locations cause errors to be returned.

The physical memory is the real memory attached to the Q-Bus or S-Bus. The logical memory is the memory as seen through the memory management unit. The I/O page is handled specially, and can only be accessed via /dev/kmem.

Examining and patching device registers is likely to lead to unexpected results when read-only or write-only bits are present. Of course Unix may crash if you interfere with the device driver.

LOGICAL MEMORY

The 68000 generates 24 bit addresses. These are split into a 4 bit segment number and a 20 bit segment offset. So we have 16 segments of 1 mbyte. User and system space share these segments. The segments are used for the following purpose:

0 Segment 0 (SYSTEXT) contains the exception vectors and the system code.

1 Segment 1 (SYSDATA) contains the systems data structures, e.g. the buffer pool.

2 Segment 2 (SYSUSER) contains the so called u-structure of UNIX plus the system stack. Whereas SYSTEXT and SYSDATA never change, segment SYSUSER is changed during each context switch.

3 Segment 3 (SYSRDR) is used by the kernel to map into arbitrary physical memory for reading.

4 Segment 4 (SYSWRT) is used by the kernel to map into arbitrary physical memory for writing. E.g. during a fork a process has to be copied in physical memory. This is done by mapping SYSRDR to the old process and SYSWRT to the new memory and copying from SYSRDR to SYSWRT.

5 Segment 5 (SYSSPCL) maps physical 0x300000 - 0x3fffff to logical 0x500000 to 0x5fffff. Via this segment the bitmap memory (in the nonvirtual version), the ethernet buffer and the floating point board are addressed. Unfortunately, this presents a security risk, because each process can at random overwrite this segment. Memory put in physical segment 3 could be addressed by all processes and serve as shared memory for special applications.

6 Segment 6 (USRTEXT) contains the user program's text segment. The text segment can extend into higher segments.

7 Segment 7 may contain more text.

8 Segment 8 (USRDATA) may contain even more text, but normally contains the user program's data segment. The data segment contains the data and bss sections of the program (see *a.out*(5)).

9 Segment 9 to segment 12 may contain additional user text, or segment 9 to segment 13 may contain additional user data.

14 Segment 14 contains the user stack. The stack grows downwards from 0xeffffff and may extend into lower segments.

The rules for user segment allocation are as follows: let btos(x) be a function that returns for a number of bytes the number of required segments, i.e. btos(x) = (x + 0xfffff) / 0x100000. Then, btos(user_code) + btos(user_data) + btos(user_stack) must be <= 9. Additionally, btos(user_data) + btos(user_stack) must be <= 7. User code starts at 0x600000. User data starts at 0x800000, unless user code is larger than 2 mbytes. Then user data starts at the next segment boundary. User stack starts at segment 14 and may extend into lower segments, but not into a segment already occupied by user data.

EXAMPLE

```
/* read location 400 - 500 of logical memory */
int m = open("/dev/kmem",2);
lseek(m,400L,0);
read(m,buf,100);

/* write a 1 into device register at 0xfffc00 */
short x;
lseek(m,0xfffc00,0);
x = 1;
write(m,&x,2);
```

FILES

/dev/mem, /dev/kmem

NAME

null – the null file

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

FILES

/dev/null

NAME

ot, ox – TM 503/TM 603/TM 703 disk, TM 100-4 floppy disk

DESCRIPTION

The octal representation of the minor device number is encoded *dp*, where *d* is a physical drive number, and *p* is a logical unit (subsection) within a physical unit. The origins and sizes of the logical units on each drive, counted in 512-byte blocks, for the TM 603 are:

log. unit	start blo.	size (in blocks)	
0	0	9600	(root on TM 603)
1	9600	4000	(swap on TM 603)
2	13600	11132	(/usr on TM 603)
3	0	24732	(all of TM 603)
4	0	12366	(1/2 of TM 603)
5	12366	12366	(1/2of TM 603)
6	0	1280	(all of TM 100-4 floppy 0)
7	0	1280	(all of TM 100-4 floppy 1)

The logical units for the TM 503 are:

log. unit	start blo.	size (in blocks)	
0	0	9600	(root on TM 503)
1	9600	4000	(swap on TM 503)
2	13600	19340	(/usr on TM 503)
3	0	32940	(all of TM 503)
4	0	16470	(1/2 of TM 503)
5	16470	16470	(1/2of TM 503)
6	0	1280	(all of TM 100-4 floppy 0)
7	0	1280	(all of TM 100-4 floppy 1)

The logical units for the TM 703 are:

log. unit	start blo.	size (in blocks)	
0	0	9600	(root on TM 703)
1	9600	4000	(swap on TM 703)
2	13600	38330	(/usr on TM 703)
3	0	51930	(all of TM 703)
4	0	25965	(1/2 of TM 703)
5	25965	25965	(1/2of TM 703)
6	0	1280	(all of TM 100-4 floppy 0)
7	0	1280	(all of TM 100-4 floppy 1)

Systems distributed for these devices use disk 0 for the root, disk 1 for swapping, and disk 3 or disk 4 or disk 5 for a mounted user file system.

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records.

A 'raw' interface provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.' In raw I/O the buffer must begin on a word boundary.

FILES
/dev/ot?, /dev/rot?
/dev/ox?, /dev/rox?

SEE ALSO
format (8)

NAME

pipe – Pipes and named pipes - everything you always wanted to know about ...

DESCRIPTION

Normal pipes are anonymous. They are created with *pipe*(2) by a process which then normally forks. The pipe system call returns two file descriptors, one for read and one for write. These file descriptors are inherited by the forked processes. Each process normally closes the file descriptor it will not use. The pipe is only accessible via file descriptors, not names, thus processes communicating over a pipe need a common ancestor which created the pipe for them.

Named pipes, on the contrary, are created as filestore files. This is done with the system call *mknod*(2) where the first parameter is a file name, and the second parameter is the mode of the file. The mode must be the bitwise or of S_IFIFO (defined in <sys/stat.h>) and the access bits, e.g.

```
mknod("np0",S_IFIFO | 0666);
```

The pipe in this example has the name np0 and read and write permission for owner, group and others. The creation can also be done with the command *mknod*(8) like this:

```
/etc/mknod np0 p
```

The pipe can now be used like a normal file. The following two commands have the same effect:

```
ls | wc

ls > np0 & wc < np0
```

The length of a write to a pipe may not exceed 5120 bytes, otherwise errno EIO will be returned. A read of a pipe when no process has the pipe open for writing will return a read count of 0. A write to a pipe when no process has the pipe open for reading will cause signal SIGPIPE and return errno EPIPE. A read of a named pipe after an open with parameter FNDELAY (see *open*(2)) will not wait if the named pipe is empty, but will return with a read count of 0 instead.

The command "ls -l" will indicate a named pipe with a p in the first column. Named pipes can be found with "find <path> -type p ...".

NAME

rk − RK-11/RK03 or RK05 disk

DESCRIPTION

Rk? refers to an entire disk as a single sequentially-addressed file. Its 256-word blocks are numbered 0 to 4871. Minor device numbers are drive numbers on one controller.

The *rk* files discussed above access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RK files begin with *rrk* and end with a number which selects the same disk as the corresponding *rk* file.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/rk?, /dev/rrk?

BUGS

In raw I/O *read* and *write*(2) truncate file offsets to 512-byte block boundaries, and *write* scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, *read*, *write* and *lseek*(2) should always deal in 512-byte multiples.

NAME

rl, hl – RL01/RL02 moving-head disk

DESCRIPTION

The octal representation of the minor device number is encoded *dp*, where *d* is a physical drive number, and *p* is a logical unit (subsection) within a physical unit. The origins and sizes of the logical units on each drive, counted in cylinders (and heads) of 20 512-byte blocks, are:

log. unit	start cyl.	length	size (in blocks)	
0	0	400	8000	(root on RL01/02)
1	400	112	2240	(swap on RL02)
2	400	107	2140	(swap on RL01)
3	0	512	10240	(first half of RL02)
4	512	507	10140	(rest of RL02)
5	0	1019	20380	(all of RL02)
6	0	507	10140	(all of RL01)
7	0	0	0	(spare)

Systems distributed for these devices use disk 0 for the root, disk 1 for swapping, and disk 6 (RL01) or disk 4 (RL02) for a mounted user file system.

The embedded bad sector replacement mechanism assumes a bad sector file on each disk pack. Before using any disk packs please check them for bad sectors with the CADMUS disk check program. For the root filesystem and the swap area we suggest to use only *bad-sector-free* disks. Otherwise the system also works well but it will slow down. If you have bad sectors in the root filesystem area you cannot boot **MUNIX** directly from the *Minitor*. The Minitor doesn't handle bad sectors. In this case you have to boot from any other device (i.e. floppy, streamer ...). Only the standalone driver and the MUNIX-driver can handle bad sectors.

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records.

A 'raw' interface provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.' In raw I/O the buffer must begin on a word boundary.

Under **MUNIX** disk volumes may be changed without rebooting the system. In standalone mode however you have to restart the standalone program after changing a disk volume.

FILES

/dev/rl?, /dev/rrl?
/dev/hl?, /dev/rhl?

SEE ALSO

format(8), check(8), iopage(7)
Bad Sector Handling (Vol. 2c)

DIAGNOSTICS

Sector numbers in error diagnostics are absolute. The sector numbers range from 0 to 40960 for an RL02 and from 0 to 20480 for an RL01. The last 200 sectors are not visible to the user. They are reserved for bad sector handling. The message *Cannot read bad sector file* using a disk pack is fatal. Do not use this disk any longer. The message *No replace sector available* indicates a new bad sector. Use the printed sector number to update the bad sector information by the disk check program.

NAME

rm – RM02/03/05 moving-head disk

DESCRIPTION

The octal representation of the minor device number is encoded *dp*, where *d* is a physical drive number, and *p* is a logical unit (subsection) within a physical unit. The origins and sizes of the logical units on each RM02/03 drive, counted in cylinders of 160 512-byte blocks and tracks of 32 512-byte blocks, are:

logical unit	starting cylinder	length (cyl.+tracks)	size (in blocks)	
0	0	60+0	9600	(root on RM02/03)
1	60	55+0	8800	(swap on RM02/03)
2	115	50+0	8000	(/sys on RM02/03)
3	165	657+0	105120	(rest of RM02/03)
4	0	0+0	0	(spare)
5	0	0+0	0	(spare)
6	0	0+0	0	(spare)
7	0	822+0	131520	(all of RM02/03)

The origins and sizes of the logical units on each RM05 drive, counted in cylinders of 608 512-byte blocks and tracks of 32 512-byte blocks, are:

logical unit	starting cylinder	length (cyl.+tracks)	size (in blocks)	
0	0	16+0	9728	(root on RM05)
1	16	14+0	8512	(swap on RM05)
2	30	14+0	8512	(/sys on RM05)
3	44	778+14	473024	(rest of RM05)
4	0	411+0	249888	(1st part of RM05)
5	411	206+0	125248	(2nd part of RM05)
6	617	205+14	125088	(last part of RM05)
7	0	822+14	500224	(all of RM05)

Systems distributed for these devices use disk 0 for the root, disk 1 for swapping, and disk 3 for a mounted user file system.

The embedded bad sector replacement mechanism assumes a bad sector file on each disk volume. Before using any disk volums please check them for bad sectors with the CADMUS disk check program. For the root filesystem and the swap area we suggest to use only *bad-sector-free* disks. Otherwise the system also works well but it will slow down. If you have bad sectors in the root filesystem area you cannot boot **MUNIX** directly from the *Minitor*. The Minitor doesn't handle bad sectors. In this case you have to boot from an other device (i.e. floppy, streamer ...). Only the standalone driver and the MUNIX-driver can handle bad sectors.

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records.

A 'raw' interface provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.' In raw I/O the buffer must begin on a word boundary.

Under **MUNIX** disk volumes may be changed without rebooting the system. In standalone mode however you have to restart the standalone program after changing a disk volume.

FILES

/dev/rm?, /dev/rrm?

SEE ALSO

hp(4), format(8)

DIAGNOSTICS

Sector numbers in error diagnostics are absolute. The sector numbers range from 0 to 131680 for RM02/03 and from 0 to 500384 for an RM05. The last 160 sectors are not visible to the user. They are reserved for bad sector handling. The message *Cannot read bad sector file* using a disk volume is fatal. Do not use this disk any longer. The message *No replace sector available* indicates a new bad sector. Use the printed sector number to update the bad sector information by the disk check program.

NAME

rx – RX01 or RX02 floppy disk

DESCRIPTION

The *rx* driver supports both double sided (DS) or single sided (SS) drives and double density (DD) or single density (SD) format.

Rx? refers to an entire disk as a single sequentially-addressed file. Its 256-word blocks are numbered 0 to (number of blocks – 1). For the number of blocks on a floppy disk see the following table:

500	SS/SD
1000	DS/SD
1001	SS/DD
2002	DS/DD

Rx0 refer to drive 0, single density format, *rx1* to drive 1, single density format, *rx2* to drive 0, double density format and *rx3* refer to drive 1, double density format. The names *rx4* to *rx7* would refer to a second controller, drive 2 and 3, if existent.

The *rx* files discussed above access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RX files begin with *rrx*.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block).

An ioctl-command, as described in /usr/include/sys/rxcmd.h, may be given to reformat a disk, swap bytes, switch sector interleave off, or modify the way in which the two sides of a double sided disks are accessed.

FILES

/dev/rx?, /dev/rrx?

SEE ALSO

rxctrl (1), format (8)

NAME

sbp – Simplified Basic Block Port Interface

SYNOPSIS

#include <sys/sbp.h>

DESCRIPTION

The *sbp* driver is a very simple driver which is derived from the driver for the Cambridge ring, see *bbp*(4).

On a ring, it is possible to send data between processes running on any machine. Specifically it is possible that the processes are running on the same machine. In this case, the data travels simply around the whole ring. But using the net for simple interprocess communication between processes running on one machine is a bit expensive. So the bbp driver has been simplified to run without net hardware, giving the sbp driver.

The sbp uses the concept of "ports". A process can open a port and specify an input and output port number. If the output port number of process A is equal to the input port number of process B, then A can send a message via the port to B. All input port numbers must be unique, whereas several processes may have the same output port number. Additional ioctl-commands allow a process to see if a message has been delivered correctly to a port, or to see if a message is waiting to be read, thus preventing blocking reads.

There exists a limited number of port structures in the kernel, e.g. 50, and a corresponding number of special files, e.g. /dev/sbp0 to /dev/sbp49. A port must be opened and then configured to be used. It is opened by repeated attempts to open /dev/sbp0, /dev/sbp1, ... until the open succeeds. The following procedure does the job:

```
#include    <errno.h>
sbopen(mode)
int mode;
{
/*
    This procedure opens the first simple block port it can find
    available, and returns the file descriptor. The port is opened
    using "mode". If an error arises -1 will be returned.
    The error EACCES indicates that the port is already in use and is
    not returned to the user.
*/
    static char PORT[] = "/dev/sbp00";
    register int i, fd;
    extern int errno;

    i = 0;
    do
    {
        if (i<10)
        {
            PORT[8]=i+'0';
            PORT[9]=' ';
        }
        else
        {
            PORT[8]= (i/10)+'0';
            PORT[9]= (i%10)+'0';
            PORT[10]=' ';
        }
        i++;
    }
    while ((fd=open(PORT,mode)) < 0 && errno==EACCES);
    return(fd);
}
```

After the open the port must be configured:

```
#include <sys/sbp.h>

struct portinfo pi;
struct portenq pe;

        fd = sbopen(2); /* open port for read and write */
        pi.pi_inport = my_own_unique_port_number;
        pi.pi_outport = his_own_unique_port_number;

        ioctl(fd,BBPSET,&pi);
```

The system will generate a unique port number if the special value DYNAMIC is assigned to pi_inport. The generated number can be obtained with BBPGET:

```
        pi.pi_inport = DYNAMIC;
        pi.pi_outport = his_own_unique_port_number;
        ioctl(fd,BBPSET,&pi);
        ioctl(fd,BBPGET,&pi);
        my_own_unique_port_number = pi.pi_inport;
```

Now we can transfer messages. Reading and writing is unbuffered and tightly interlocked. The read and write count may not exceed the system buffer size SBUFSIZE (see /usr/include/sys/param.h). When a write is done, the driver will grab a buffer from the systems disk buffer pool, copy the user data to it and hook the buffer to the destination port. If another block is already hooked to this port, the writing process will be delayed until the other block has been read. When a read is done, the read will be delayed until a block is hooked to the port. The data will then be copied from the system buffer to the user buffer and the block will be returned to the buffer pool.

The ioctl call BBPENQ returns a structure with three fields:

- The field *pn_sendport* contains the sender port number of the last received block.
- The field *pn_xrslt* contains the value BB_ACCEPTED, if the last write was successful, BB_ABSENT if the output port number was not the input port number of any port, or if this port was not open for reading.
- The field *pn_blkavail* is non-zero, if a block is waiting to be read on this port. This allows to implement non-blocking reads.

The following example illustrates reading and writing:

```
        char buf[BUFSIZ];

        ioctl(fd,BBPENQ,&pe);
        while (!pe.pn_blkavail)
                sleep(5);
        len = read(fd,buf,BUFSIZ);
        ioctl(fd,BBPENQ,&pe);
        sender = pe.pn_sendport;

        write(fd,buf,cnt);
        ioctl(fd,BBPENQ,&pe);
        if (pe.pn_xrslt == BB_ABSENT)
```

```
printf("destination %d has gone\n",pi.pi_outport);
```

NAME

st − SCT11 Streamer interface

DESCRIPTION

The special files *rst0*, *nrst0* refer to the streamer drive 0. The letter r indicates "raw" device, the letter n indicates "no rewind" when the streamer is closed. For *nrst0* the bit 0200 in the minor device number must be set. To rewind the streamer tape you can say **< /dev/rst0**.

The raw devices *rst0* and *nrst0* allow transfers of up to 127 512-byte blocks with a single *read* or *write* call. The byte count should always be an exact multiple of 512.

There is an *ioctl(2)* call applying to the streamer. The *ioctl* call has the form:

```
#include <sys/stcmd.h>
ioctl (fildes, command)
```

The applicable commands are:

ERCMD
: Erase the contents of the tape and rewind it.

RETCMD
: Make a retension of the tape and rewind it.

CBUF
: Switch the streamer driver to work in **double buffer mode.** This mode allows asynchronous I/O-transfers to and from the streamer. The first *read* or *write* call after the *ioctl* call initiates only the I/O-transfer and reports no error. A succeeding *read/write* call waits until the current transfer is finished, reports the errors and initiates the new transfer. A *read/write* call with byte count 0 doesn't initiate a new transfer. It waits only for the end of the previous one and reports the errors.
While data is transferred to/from the streamer any other operation can be done asynchronously. For example a second I/O-buffer may be filled from the disk and afterwards written to the streamer.
The double buffer mode is primarily used for physical disk back-up (*stvolcopy*, *stcp*) to keep the tape streaming. A *close* call switches back to normal streamer mode.

REMARKS

A physical copy from disk to tape is done very quickly. Using the program *stvolcopy* while no other users are on the system, **MUNIX** feeds the streamer with data at the highest possible speed. Thus the tape keeps streaming.
Unfortunately, when making a logical back-up from disk to tape, the streamer needs data faster than **MUNIX** can provide. Therefore, the streamer over- or underruns its internal buffers and does not stream. This means that after it has transferred some data, the tape will stop. When subsequent data arrives, it will rewind a piece of tape, then read forward to where it stopped, and immediately write the data. This zigzag-motion of the tape can be very time-consuming. The "zig" occurs when transferring data, and its time is proportional to the amount of

stvolcopy geht NICHT [illegible]

data transferred. The "zag" is the rewind, and its time is constant. The ratio of "zig"-time to "zag"-time becomes tolerable, when the amount of data transferred with one "zig" gets large.

Three programs have been modified to use larger buffersizes when working with the streamer: *cpio*, *volcopy* and *cp*. The *volcopy* for streamer has been renamed *stvolcopy*, the *cp* has been renamed *stcp*. Both programs use double buffering. For *cpio* the option –S sets the blocking factor to 120, but does not invoke double buffering. The standalone program *volcopy* in /sa1 or /sa2 can copy in tapes written with *stvolcopy*.
Stcp is intended to write standalone programs onto the streamer tape.

These programs allow you to backup your disks properly. Once in a while you should make physical copies of your file systems with *stvolcopy*. *Stvolcopy* can then be used to recreate a file system after a catastrophic failure. The standalone version of *volcopy* is needed to recreate the root filesystem.
At least once a day you should make an incremental dump of all files with *cpio*. From the cpio-tapes you can easily retrieve single files or whole directories.

EXAMPLES

incremental dump (last three days, say)

```
find / -mtime -3 -print | cpio -oS >/dev/rst0
```

file retrieval:

```
cpio -ivSmd myfile </dev/rst0
```

physical dump:

```
           special   fsname  volume
labelit /dev/rst0 root tape1 -n
stvolcopy root /dev/rhk0 hk0 /dev/rst0 tape1
          fsname  special 1  volname1  special 2  volname 2
```

standalone (only terminal input shown):

```
/sa1/volcopy
g0
-S root st(0,0) tape1 hk(0,0) root
```

write standalone programs onto tape

```
< /dev/rst0
stcp /sa/boot /dev/nrst0
stcp /sa/volcopy /dev/nrst0
stcp /sa/check /dev/rst0
```

SEE ALSO

volcopy(8), cpio(1), labelit(8), stctrl(1), cp(1)
ioctl(2), read(2), write(2)

BUGS

Do not care too much for the message *streamer error: cannot read status*. In most cases the last streamer operation was completed sucessfully.

NAME

termio − general terminal interface

DESCRIPTION

All of the asynchronous communications ports use the same general interface, no matter what hardware is involved. This section discusses the common features of this interface.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by *getty*(8) and become a user's standard input, output, and error files. The very first terminal file opened by the process group leader of a terminal file not already associated with a process group becomes the *control terminal* for that process group. The control terminal plays a special role in handling quit and interrupt signals, as discussed below. The control terminal is inherited by a child process during a *fork*(2). A process can break this association by changing its process group using *setpgrp*(2).

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely full, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently, this limit is 256 characters. When the input limit is reached, all the saved characters are thrown away without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a new-line (ASCII LF) character, an end-of-file (ASCII CTRL-Z) character, or an end-of-line character. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, erase and kill processing is normally done. By default, the character BS or DEL erases the last character typed, except that it will not erase beyond the beginning of the line. By default, the character CTRL-X kills (deletes) the entire input line, and optionally outputs a new-line character. Both these characters operate on a key-stroke basis, independently of any backspacing or tabbing that may have been done. Both the erase and kill characters may be entered literally by preceding them with the escape character (\). In this case the escape character is not read. The erase and kill characters may be changed.

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

INTR (Control-c) generates an *interrupt* signal which is sent to all processes with the associated control terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see *signal*(2).

QUIT	(Control-y) generates a *quit* signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called core) will be created in the current working directory.
ERASE	(Backspace or DEL) erases the preceding character. It will not erase beyond the start of a line, as delimited by a NL, EOF, or EOL character.
KILL	(Control-x) deletes the entire line, as delimited by a NL, EOF, or EOL character.
EOF	(Control-z) may be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a newline, and the EOF is discarded. Thus, if there are no characters waiting, which is to say the EOF occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.
NL	(ASCII LF) is the normal line delimiter. It can not be changed or escaped.
EOL	(ASCII NUL) is an additional line delimiter, like NL. It is not normally used.
STOP	(Control-s or ASCII DC3) can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.
START	(Control-q or ASCII DC1) is used to resume output which has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The start/stop characters can not be changed or escaped.

The character values for INTR, QUIT, ERASE, KILL, EOF, and EOL may be changed to suit individual tastes. The ERASE, KILL, and EOF characters may be escaped by a preceding \ character, in which case no special function is done.

When the carrier signal from the data-set drops, a *hangup* signal is sent to all processes that have this terminal as the control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hangup signal is ignored, any subsequent read returns with an end-of-file indication. Thus programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. If a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold, the program is resumed.

Several *ioctl*(2) system calls apply to terminal files. The primary calls use the following structure, defined in <termio.h>:

```
#define NCC             8
struct  termio {
        unsigned short  c_iflag;                /* input modes */
        unsigned short  c_oflag;                /* output modes */
        unsigned short  c_cflag;                /* control modes */
        unsigned short  c_lflag;                /* local modes */
        char            c_line;                 /* line discipline */
        unsigned char   c_cc[NCC];          /* control chars */
};
```

The special control characters are defined by the array *c_cc*. The relative positions and initial values for each function are as follows:

0	INTR	Ctrl-c
1	QUIT	Ctrl-y
2	ERASE	BS
3	KILL	Ctrl-x
4	EOF	Ctrl-z
5	EOL	NUL
6	reserved	
7	reserved	

The *c_iflag* field describes the basic terminal input control:

IGNBRK	0000001	Ignore break condition.
BRKINT	0000002	Signal interrupt on break.
IGNPAR	0000004	Ignore characters with parity errors.
PARMRK	0000010	Mark parity errors.
INPCK	0000020	Enable input parity check.
ISTRIP	0000040	Strip character.
INLCR	0000100	Map NL to CR on input.
IGNCR	0000200	Ignore CR.
ICRNL	0000400	Map CR to NL on input.
IUCLC	0001000	Map upper-case to lower-case on input.
IXON	0002000	Enable start/stop output control.
IXANY	0004000	Enable any character to restart output.
IXOFF	0010000	Enable start/stop input control.

If IGNBRK is set, the break condition (a character framing error with data all zeros) is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise if BRKINT is set, the break condition will generate an interrupt signal and flush both the input and output queues. If IGNPAR is set, characters with other framing and parity errors are ignored.

If PARMRK is set, a character with a framing or parity error which is not ignored is read as the three character sequence: 0377, 0, X, where X is the data of the character received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid character of 0377 is read as 0377, 0377. If PARMRK is not set, a framing or parity error which is not ignored is read as the character NUL (0).

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled. This allows output parity generation without input parity errors.

If ISTRIP is set, valid input characters are first stripped to 7-bits, otherwise all 8-bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). Otherwise if ICRNL is set, a received CR character is translated into a NL character.

If IUCLC is set, a received upper-case alphabetic character is translated into the corresponding lower-case character.

If IXON is set, start/stop output control is enabled. A received STOP character will suspend output and a received START character will restart output. All start/stop characters are ignored and not read. If IXANY is set, any input character, will restart output which has been suspended.

If IXOFF is set, the system will transmit START/STOP characters when the input queue is nearly empty/full.

The initial input control value is all bits clear.

The *c_oflag* field specifies the system treatment of output:

OPOST	0000001	Postprocess output.
OLCUC	0000002	Map lower case to upper on output.
ONLCR	0000004	Map NL to CR-NL on output.
OCRNL	0000010	Map CR to NL on output.
ONOCR	0000020	No CR output at column 0.
ONLRET	0000040	NL performs CR function.
OFILL	0000100	Use fill characters for delay.
OFDEL	0000200	Fill is DEL, else NUL.
NLDLY	0000400	Select new-line delays:
NL0	0	
NL1	0000400	
CRDLY	0003000	Select carriage-return delays:
CR0	0	
CR1	0001000	
CR2	0002000	
CR3	0003000	
TABDLY	0014000	Select horizontal-tab delays:
TAB0	0	
TAB1	0004000	
TAB2	0010000	
TAB3	0014000	Expand tabs to spaces.
BSDLY	0020000	Select backspace delays:
BS0	0	
BS1	0020000	
VTDLY	0040000	Select vertical-tab delays:
VT0	0	
VT1	0040000	
FFDLY	0100000	Select form-feed delays:
FF0	0	
FF1	0100000	

If OPOST is set, output characters are post-processed as indicated by the remaining flags, otherwise characters are transmitted without change.

If OLCUC is set, a lower-case alphabetic character is transmitted as the corresponding upper-case character. This function is often used in conjunction with IUCLC.

If ONLCR is set, the NL character is transmitted as the CR-NL character pair. If OCRNL is set, the CR character is transmitted as the NL character. If ONOCR is set, no CR character is transmitted when at column 0 (first position). If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer will be set to 0 and the delays specified for CR will be used. Otherwise the NL character is assumed to do just the line-feed function; the column pointer will remain unchanged. The column pointer is also set to 0 if the CR character is actually transmitted.

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay. If OFILL is set, fill characters will be transmitted for delay instead of a timed delay. This is useful for high baud rate terminals which need only a minimal delay. If OFDEL is set, the fill character is DEL, otherwise NUL.

If a form-feed or vertical-tab delay is specified, it lasts for about 2 seconds.

New-line delay lasts about 0.10 seconds. If ONLRET is set, the carriage-return delays are used instead of the new-line delays. If OFILL is set, two fill characters will be transmitted.

Carriage-return delay type 1 is dependent on the current column position, type 2 is about 0.10 seconds, and type 3 is about 0.15 seconds. If OFILL is set, delay type 1 transmits two fill characters, and type 2 four fill characters.

Horizontal-tab delay type 1 is dependent on the current column position. Type 2 is about 0.10 seconds. Type 3 specifies that tabs are to be expanded into spaces. If OFILL is set, two fill characters will be transmitted for any delay.

Backspace delay lasts about 0.05 seconds. If OFILL is set, one fill character will be transmitted.

The actual delays depend on line speed and system load.

The initial output control value is all bits clear.

The *c_cflag* field describes the hardware control of the terminal:

CBAUD	0000017	Baud rate:
B0	0	Hang up
B50	0000001	50 baud
B75	0000002	75 baud
B110	0000003	110 baud
B134	0000004	134.5 baud
B150	0000005	150 baud
B200	0000006	200 baud
B300	0000007	300 baud
B600	0000010	600 baud
B1200	0000011	1200 baud

B1800	0000012	1800 baud
B2400	0000013	2400 baud
B4800	0000014	4800 baud
B9600	0000015	9600 baud
EXTA	0000016	External A
EXTB	0000017	External B
CSIZE	0000060	Character size:
CS5	0	5 bits
CS6	0000020	6 bits
CS7	0000040	7 bits
CS8	0000060	8 bits
CSTOPB	0000100	Send two stop bits, else one.
CREAD	0000200	Enable receiver.
PARENB	0000400	Parity enable.
PARODD	0001000	Odd parity, else even.
HUPCL	0002000	Hang up on last close.
CLOCAL	0004000	Local line, else dial-up.

The CBAUD bits specify the baud rate. The zero baud rate, B0, is used to hang up the connection. If B0 is specified, the data-terminal-ready signal will not be asserted. Normally, this will disconnect the line. For any particular hardware, impossible speed changes are ignored.

The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used, otherwise one stop bit. For example, at 110 baud, two stops bits are required.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, the PARODD flag specifies odd parity if set, otherwise even parity is used.

If CREAD is set, the receiver is enabled. Otherwise no characters will be received.

If HUPCL is set, the line will be disconnected when the last process with the line open closes it or terminates. That is, the data-terminal-ready signal will not be asserted.

If CLOCAL is set, the line is assumed to be a local, direct connection with no modem control. Otherwise modem control is assumed.

The initial hardware control value after open is B300, CS8, CREAD, HUPCL.

The *c_lflag* field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline (0) provides the following:

ISIG	0000001	Enable signals.
ICANON	0000002	Canonical input (erase and kill processing).
XCASE	0000004	Canonical upper/lower presentation.
ECHO	0000010	Enable echo.
ECHOE	0000020	Echo erase character as BS-SP-BS.
ECHOK	0000040	Echo NL after kill character.
ECHONL	0000100	Echo NL.
NOFLSH	0000200	Disable flush after interrupt or quit.

If ISIG is set, each input character is checked against the special control characters INTR and QUIT. If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus these special input functions are possible only if ISIG is set. These functions may be disabled individually by changing the value of the control character to an unlikely or impossible value (e.g. 0377).

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL. If ICANON is not set, read requests are satisfied directly from the input queue. A read will not be satisfied until at least MIN characters have been received or the timeout value TIME has expired. This allows fast bursts of input to be read efficiently while still allowing single character input. The MIN and TIME values are stored in the position for the EOF and EOL characters respectively. The time value represents tenths of seconds.

If XCASE is set, and if ICANON is set, an upper-case letter is accepted on input by preceding it with a \ character, and is output preceded by a \ character. In this mode, the following escape sequences are generated on output and accepted on input:

for:	*use*:
`	\'
|	\!
~	\^
{	\(
}	\)
\	\\

For example, A is input as \a, \n as \\n, and \N as \\\n.

If ECHO is set, characters are echoed as received.

When ICANON is set, the following echo functions are possible. If ECHO and ECHOE are set, the erase character is echoed as ASCII BS SP BS, which will clear the last character from a CRT screen. If ECHOE is set and ECHO is not set, the erase character is echoed as ASCII SP BS. If ECHOK is set, the NL character will be echoed after the kill character to emphasize that the line will be deleted. Note that an escape character preceding the erase or kill character removes any special function. If ECHONL is set, the NL character will be echoed even if ECHO is not set. This is useful for terminals set to local echo (so-called half duplex). Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

If NOFLSH is set, the normal flush of the input and output queues associated with the quit and interrupt characters will not be done.

The initial line-discipline control value is all bits clear.

The primary *ioctl*(2) system calls have the form:

```
ioctl (fildes, command, arg)
struct termio *arg;
```

The commands using this form are:

TCGETA	Get the parameters associated with the terminal and store in the *termio* structure referenced by arg.
TCSETA	Set the parameters associated with the terminal from the structure referenced by arg. The change is immediate.
TCSETAW	Wait for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.
TCSETAF	Wait for the output to drain, then flush the input queue and set the new parameters.

Additional *ioctl*(2) calls have the form:

```
ioctl (fildes, command, arg)
int arg;
```

The commands using this form are:

TCSBRK	Wait for the output to drain. If *arg* is 0, then send a break (zero bits for 0.25 seconds).
TCXONC	Start/stop control. If *arg* is 0, suspend output; if 1, restart suspended output.
TCFLSH	If *arg* is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

FILES

/dev/tty
/dev/tty*
/dev/console

SEE ALSO

stty(1), ioctl(2).

NAME

tm,ts – TM-11/TU-10 magtape interface, TS-11 magtape interface

DESCRIPTION

The files *mt0,nmt0,rmt0,nrmt0* refer to the DEC TU10/TM11 resp. TS-11 magtape. When closed it can be rewound or not, see below. If it was open for writing, two end-of-files are written. If the tape is not to be rewound it is positioned with the head between the two tapemarks.

If the 0200 bit is on in the minor device number the tape is not rewound when closed. The names for these files begin with the letter n for "no rewind": nmt0, nrmt0.

A standard tape (with name mt0 or nmt0) consists of a series of 512 byte records terminated by an end-of-file. To the extent possible, the system makes it possible, if inefficient, to treat the tape like any other file. Seeks have their usual meaning and it is possible to read or write a byte at a time. Writing in very small units is inadvisable, however, because it tends to create monstrous record gaps.

The *mt* files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the 'raw' interface is appropriate. The associated files are named *rmt0, nrmt0.* Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, provided it is no greater than the buffer size; if the record is long, an error is indicated. In raw tape I/O, the buffer must begin on a word boundary and the count must be even. Seeks are ignored. A zero byte count is returned when a tape mark is read, but another read will fetch the first record of the new tape file.

It is possible to skip records or files back- and forward. For compatibility with tapes written on a PDP or VAX, it is possible to instruct the driver to swap the bytes in a word. The command *mt*(1) can be called for these operations, or the ioctl call described in */usr/include/sys/mtio.h* can be used from within a program.

NAME

tty – controlling terminal interface

DESCRIPTION

The file **/dev/tty** is, in each process, a synonym for the control terminal associated with the process group of that process, if any. It is useful for programs or shell sequences that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand the name of a file for output, when typed output is desired and it is tiresome to find out what terminal is currently in use.

FILES

/dev/tty

NAME

vp – Versatec printer-plotter

DESCRIPTION

Vp is the interface to a Versatec V80 printer-plotter with a Versatec Q-Bus controller. Ordinarily bytes written on it are interpreted as ASCII characters and printed. As a printer, it writes 64 lines of 132 characters each on 11 by 8.5 inch paper. Only some of the ASCII control characters are interpreted.

NL performs the usual new-line function, i.e. spaces up the paper and resets to the left margin. It is ignored however following a CR which ends a non-empty line.

CR is ignored if the current line is empty but is otherwise like NL.

FF resets to the left margin and then to the top of the next page.

EOT resets to the left margin, advances 8 inches, and then performs a FF.

The *ioctl*(2) system call described in */usr/include/sys/vcmd.h* may be used to change the mode of the device. Only the first word of the 3-word argument structure is used. The bits mean:

0400 (VPRINTPLOT)
Enter simultaneous print/plot mode.

0200 (VPLOT)
Enter plot mode.

0100 (VPRINT)
Enter print mode (default on open).

On open a reset, clear, and form-feed are performed automatically. Notice that the mode bits are not encoded, so that it is required that exactly one be set.

In plot mode each byte is interpreted as 8 bits of which the high-order is plotted to the left; a '1' leaves a visible dot. A full line of dots is produced by 264 bytes; lines are terminated only by count or by a remote terminate function. There are 200 dots per inch both vertically and horizontally.

When simultaneous print-plot mode is entered exactly one line of characters, terminated by NL, CR, or the remote terminate function, should be written. Then the device enters plot mode and at least 20 lines of plotting bytes should be sent. As the line of characters (which is 20 dots high) is printed, the plotting bytes overlay the characters. Notice that it is impossible to print characters on baselines that differ by fewer than 20 dot-lines.

In print mode lines may be terminated either with an appropriate ASCII character or by using the remote terminate function.

FILES

/dev/vp

NAME

intro – introduction to file formats

DESCRIPTION

This section outlines the formats of various files. The C **struct** declarations for the file formats are given where applicable. Usually, these structures can be found in the directories **/usr/include** or **/usr/include/sys**.

NAME

a.out − format of programs and modules

SYNOPSIS

```
#include <a.out.h>

#include <sys/seg.h>
```

DESCRIPTION

A.out is the format of program or module files, as produced by all compilers and the link editor *ld*(1). Layout information as given in the include file is:

```
struct  exec {  /* a.out header */
        short           a_magic;        /* magic number */
        long            a_text;         /* size of text segment */
        long            a_data;         /* size of initialized data */
        long            a_bss;          /* size of unitialized data */
        long            a_syms;         /* size of symbol table */
        long            a_entry;        /* entry point */
        long            a_stksiz;       /* length of stack */
        short           a_flag;         /* relocation info stripped */
};

#define         A_MAGIC1        0407            /* normal */
#define         A_MAGIC2        0410            /* read-only text */
#define         A_MAGIC3        0411            /* separated I&D */
#define         A_MAGIC4        0405            /* overlay */

#define NCPS 16 /* maximum length of external names */
struct  nlist {         /* symbol table entry */
        char            n_name[NCPS];   /* symbol name */
        short           n_type;         /* symbol type */
        long            n_value;        /* value */
};

/* values of sigth-field */
#define RSIZE   01  /* e.g. for bra.w L1 */
#define WSIZE   02  /* e.g. for jmp.w L1 */
#define LSIZE   04  /* e.g. for jmp.l L1 */

/* values for type flag */
#define         N_UNDF  0       /* undefined */
#define         N_ABS   01      /* absolute */
#define         N_TEXT  02      /* text symbol */
#define         N_DATA  03      /* data symbol */
#define         N_BSS   04      /* bss symbol */
#define         N_TYPE  037
#define         N_REG   024     /* register name */
#define         N_FN    037     /* file name symbol */
#define         N_EXT   040     /* external bit, or'ed in */
#define FORMAT  "%08lx"     /* to print a value */

/* values of relocation bits */
#define R_WORD   01
#define R_TEXT   02
#define R_DATA   04
#define R_BSS    06
#define R_UNDF  010

/*
 * Macros which take exec structures as arguments and tell whether
 * the file has a reasonable magic number or offsets to text|symbols|strings.
 */
#define N_BADMAG(x) \
    (((x).a_magic)!=A_MAGIC1 && ((x).a_magic)!=A_MAGIC2 && \
    ((x).a_magic)!=A_MAGIC3 && ((x).a_magic)!=A_MAGIC4)
```

```
#define N_TXTOFF(x)  sizeof (struct exec)
#define N_SYMOFF(x) \
        (N_TXTOFF(x) + ((x)a_rflag ? 2 : 1) * ((x).a_text+(x).a_data))
#define N_STROFF(x) \
        (N_SYMOFF(x) + (x).a_syms)
```

The file has siz sections: a header, the program text, the data text, program relocation information, data relocation information, and a symbol table (in that order). The last three may be empty if the program was loaded with the '−s' option of *ld* or if the symbols and relocation have been removed by *strip*(1).

In the header the sizes of each section are given in bytes, but are even. The size of the header is not included in any of the other sizes.

A module has the magic number 0407 in he header. When all external references are satisfied, the loader produces a program with the magic number 0411.

When a program file is loaded into memory for execution, three logical segments are set up: the text segment, the data segment (with uninitialized data, which starts off as all 0, following initialized), and a stack. With MUNIX, the text segment is always pure, write protected and shared, and moreover instruction and data space are separated; the text begins at location USRTEXT (0x600000) and the data at location USRDATA (0x800000), see *<sys/seg.h>*.

The stack will start below location USRSTCK (0xF00000) growing downwards. The stack is automatically extended if possible. It will always be possible with the Motorola 68010 CPU. With the M68000, only in some cases will the processor be able to restart an instruction that caused stack overflow. In the other cases, the stack must be enlarged using *stksiz*(1). The data segment is only extended as requested by *brk*(2).

The start of the text segment in the file is 034(8); the start of the data segment is 034+S_t (the size of the text); the start of the relocation information is 034+S_t+S_d; the start of the symbol table is 034+2(S_t+S_d) if the relocation information is present, 034+S_t+S_d if not.

The layout of a symbol table entry and the principal flag values that distinguish symbol types are given in the include file.

If a symbol's type is undefined external, and the value field is non-zero, the symbol is interpreted by the loader *ld* as the name of a common region whose size is indicated by the value of the symbol.

The value of a word in the text or data portions which is not a reference to an undefined external symbol is exactly that value which will appear in memory when the file is executed. If a word in the text or data portion involves a reference to an undefined external symbol, as indicated by the relocation information for that word, then the value of the word as stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added into the word in the file.

If relocation information is present, it amounts to one word per word of program text or initialized data. There is no relocation information if the 'relocation info present' flag in the header is off. If the code contains a long (32 bit) reference to an undefined external symbol at location x relative to the start of the text, the corresponding relocation info is in the word x relative to the start of the relocation info, and the word x+2 is 0.

Bits 3-1 of a relocation word indicate the segment referred to by the text or data word associated with the relocation word:

0 absolute number

R_TEXT
reference to text segment

R_DATA
reference to initialized data

R_BSS
reference to uninitialized data (bss)

R_UNDF
reference to undefined external symbol

Bit 0 of the relocation word indicates, if 1, that the reference is relative to the pc (e.g. 'bra.w x'); if 0, that the reference is to the actual symbol (e.g., 'jmp x').

The remainder of the relocation word (bits 15-4) contains a symbol number in the case of external references, and is unused otherwise. The first symbol is numbered 0, the second 1, etc.

SEE ALSO
ld(1), nm(1)

NAME

acct — per-process accounting file format

SYNOPSIS

```
#include <sys/acct.h>
```

DESCRIPTION

Files produced as a result of calling *acct*(2) have records in the form defined by <sys/acct.h>, whose contents are:

```
typedef ushort comp_t;  /* "floating point" */
                        /* 13-bit fraction, 3-bit exponent  */

struct  acct
{
        char    ac_flag;        /* Accounting flag */
        char    ac_stat;        /* Exit status */
        ushort  ac_uid;         /* Accounting user ID */
        ushort  ac_gid;         /* Accounting group ID */
        dev_t   ac_tty;         /* control typewriter */
        time_t  ac_btime;       /* Beginning time */
        comp_t  ac_utime;       /* acctng user time in clock ticks */
        comp_t  ac_stime;       /* acctng system time in clock ticks */
        comp_t  ac_etime;       /* acctng elapsed time in clock ticks */
        comp_t  ac_mem;         /* memory usage */
        comp_t  ac_io;          /* chars transferred */
        comp_t  ac_rw;          /* blocks read or written */
        char    ac_comm[8];     /* command name */
};

extern  struct  acct    acctbuf;
extern  struct  inode   *acctp;         /* inode of accounting file */

#define AFORK   01              /* has executed fork, but no exec */
#define ASU     02              /* used super-user privileges */
#define ACCTF   0300            /* record type: 00 = acct */
```

In *ac_flag*, the AFORK flag is turned on by each *fork*(2) and turned off by an *exec*(2). The *ac_comm* field is inherited from the parent process and is reset by any *exec*. Each time the system charges the process with a clock tick, it also adds to *ac_mem* the current process size, computed as follows:

(data size) + (text size) / (number of in-core processes using text)

The value of *ac_mem / ac_stime* can be viewed as an approximation to the mean process size, as modified by text-sharing.

The following structure represents the total accounting format used by the various accounting commands:

```
struct  tacct   {
        uid_t             ta_uid;          /* userid */
        char              ta_name[8];      /* login name */
        float             ta_cpu[2];       /* cum. cpu time, p/np (mins) */
        float             ta_kcore[2];     /* cum kcore-minutes, p/np */
        float             ta_con[2];       /* cum. connect time, p/np, mins */
        float             ta_du;           /* cum. disk usage */
        long              ta_pc;           /* count of processes */
        unsigned short    ta_sc;           /* count of login sessions */
        unsigned short    ta_dc;           /* count of disk samples */
        unsigned short    ta_fee;          /* fee for special services */
};
```

The float numbers above are in FFP (see *fp*(3)) format.

SEE ALSO

acct(8), acctcom(1), acct(2).

BUGS

The *ac_mem* value for a short-lived command gives little information about the actual size of the command, because *ac_mem* may be incremented while a different command (e.g., the shell) is being executed by the process.

NAME

ar – archive (library) file format

SYNOPSIS

#include <ar.h>

DESCRIPTION

The archive command *ar* is used to combine several files into one. Archives are used mainly as libraries to be searched by the link-editor *ld*.

A file produced by *ar* has a magic number at the start, followed by the constituent files, each preceded by a file header. The magic number and header layout as described in the include file are:

```
#define ARMAG 0177545
struct  ar_hdr {
        char   ar_name[14];
        long   ar_date;
        char   ar_uid;
        char   ar_gid;
        short  ar_mode;
        long   ar_size;
};
```

The name is a null-terminated string; the date is in the form of *time*(2); the user ID and group ID are numbers; the mode is a bit pattern per *chmod*(2); the size is counted in bytes.

Each file begins on a word boundary; a null byte is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

If the first member in a library has the name __.SYMDEF, the library has been processed by *ranlib*(1).

Notice there is no provision for empty areas in an archive file.

SEE ALSO

ar(1), ld(1), nm(1), ranlib(1)

BUGS

Coding user and group IDs as characters is a botch.

NAME

checklist – list of file systems processed by fsck

DESCRIPTION

Checklist resides in directory **/etc** and contains a list of at most 15 *special file* names. Each *special file* name is contained on a separate line and corresponds to a file system. Each file system will then be automatically processed by the *fsck*(8) command.

The name of the root file system should be a block special file, the other names can be character special files to speed things up, e.g.

```
/dev/hk0
/dev/rtmp
/dev/rhk1
/dev/rhk2
/dev/rhk3
```

SEE ALSO

fsck(8).

NAME

core – format of core image file

DESCRIPTION

UNIX writes out a core image of a terminated process when any of various errors occur. See *signal*(2) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called 'core' and is written in the process's working directory (provided it can be; normal access controls apply).

The first 3072 bytes of the core image are a copy of the system's per-user data for the process, see */usr/include/sys/user.h* for the format of this area. Then follows an area with the saved registers, in the format given in */usr/include/sys/reg.h* where the union exu has always the form ex2o, altogether VECSIZE bytes. Then follows the programs data plus the programs stack. The text segment is not dumped.

In general the debugger *adb*(1) is sufficient to deal with core images.

SEE ALSO

adb(1), signal(2)

NAME

cpio – format of cpio archive

DESCRIPTION

The *header* structure, when the -c option of *cpio*(1) is not used, is:

```
struct {
        short   h_magic,
                h_dev;
        ushort  h_ino,
                h_mode,
                h_uid,
                h_gid;
        short   h_nlink,
                h_rdev,
                h_mtime[2],
                h_namesize,
                h_filesize[2];
        char    h_name[h_namesize rounded to word];
} Hdr;
```

When the -c option is used, the *header* information is described by the statement below:

```
        sscanf(Chdr,"%6o%6o%6o%6o%6o%6o%6o%6o%11lo%6o%6o%s",
                &Hdr.h_magic,&Hdr.h_dev,&Hdr.h_ino,&Hdr.h_mode,
                &Hdr.h_uid,&Hdr.h_gid,&Hdr.h_nlink,&Hdr.h_rdev,
                &Longtime,&Hdr.h_namesize,&Longfile,Hdr.h_name);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file are recorded in an element of the array of varying length structures, *archive*, together with other items describing the file. Every instance of *h_magic* contains the constant 070707 (octal). The items *h_dev* through *h_mtime* have meanings explained in *stat*(2). The length of the null-terminated path name *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Special files, directories, and the trailer are recorded with *h_filesize* equal to zero.

SEE ALSO

cpio(1), find(1), stat(2).

NAME

dir – format of directories

SYNOPSIS

```
#include <sys/dir.h>
```

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry (see *fs*(5)). The structure of a directory entry as given in the include file is:

```
#ifndef DIRSIZ
#define DIRSIZ  14
#endif
struct  direct
{
        ino_t   d_ino;
        char    d_name[DIRSIZ];
};
```

By convention, the first two entries in each directory are for . and .. . The first is an entry for the directory itself. The second is for the parent directory. The meaning of .. is modified for the root directory of the master file system; there is no parent, so .. has the same meaning as . .

SEE ALSO

fs(5).

NAME

dump, ddate − incremental dump format

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ino.h>
#include <dumprestor.h>
```

DESCRIPTION

Tapes used by *dump*(8) and *restor*(8) contain:

a header record
two groups of bit map records
a group of records describing directories
a group of records describing files

The format of the header record and of the first record of each description as given in the include file <*dumprestor.h*> is:

```
#define NTREC           20
#define MLEN            16
#define MSIZ            4096

#define TS_TAPE         1
#define TS_INODE        2
#define TS_BITS         3
#define TS_ADDR         4
#define TS_END          5
#define TS_CLRI         6
#define MAGIC           (int)60011
#define CHECKSUM        (int)84446
struct  spcl
{
        short   c_type;
        time_t  c_date;
        time_t  c_ddate;
        short   c_volume;
        daddr_t c_tapea;
        ino_t   c_inumber;
        short   c_magic;
        short   c_checksum;
        struct  dinode  c_dinode;
        short   c_count;
        char    c_addr[BSIZE];
} spcl;

struct  idates
{
        char    id_name[16];
        char    id_incno;
        time_t  id_ddate;
};
```

NTREC is the number of 512 byte records in a physical tape block. *MLEN* is the number of bits in a bit map word. *MSIZ* is the number of bit map words.

The *TS_* entries are used in the *c_type* field to indicate what sort of header this is. The types and their meanings are as follows:

TS_TAPE	Tape volume label
TS_INODE	A file or directory follows. The *c_dinode* field is a copy of the disk inode and contains bits telling what sort of file this is.
TS_BITS	A bit map follows. This bit map has a one bit for each inode that was dumped.

TS_ADDR	A subrecord of a file description. See *c_addr* below.
TS_END	End of tape record.
TS_CLRI	A bit map follows. This bit map contains a zero bit for all inodes that were empty on the file system when dumped.
MAGIC	All header records have this number in *c_magic*.
CHECKSUM	Header records checksum to this value.

The fields of the header structure are as follows:

c_type	The type of the header.
c_date	The date the dump was taken.
c_ddate	The date the file system was dumped from.
c_volume	The current volume number of the dump.
c_tapea	The current number of this (512-byte) record.
c_inumber	The number of the inode being dumped if this is of type *TS_INODE*.
c_magic	This contains the value *MAGIC* above, truncated as needed.
c_checksum	This contains whatever value is needed to make the record sum to *CHECKSUM*.
c_dinode	This is a copy of the inode as it appears on the file system; see *filesystem*(5).
c_count	The count of characters in *c_addr*.
c_addr	An array of characters describing the blocks of the dumped file. A character is zero if the block associated with that character was not present on the file system, otherwise the character is non-zero. If the block was not present on the file system, no block was dumped; the block will be restored as a hole in the file. If there is not sufficient space in this record to describe all of the blocks in a file, *TS_ADDR* records will be scattered through the file, each one picking up where the last left off.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a *TS_END* record and then the tapemark.

The structure *idates* describes an entry of the file */etc/ddate* where dump history is kept. The fields of the structure are:

id_name	The dumped filesystem is '/dev/*id_nam*'.
id_incno	The level number of the dump tape; see *dump*(8).
id_ddate	The date of the incremental dump in system format see *types*(7).

FILES

/etc/ddate

SEE ALSO

dump(8), dumpdir(8), restor(8), filesystem(5), types(7)

NAME

/etc/map_port_eadr – table of ethernet addresses

DESCRIPTION

/etc/map_port_eadr is just a linear table of 6 byte ethernet addresses, indexed by the station number (sometimes also called identifier). The address is arbitrary for the 3COM ethernet hardware, but may be hardwired in other controllers at a later time. At any time. there must be a one-to-one correspondance between the station numbers and ethernet addresses of machines connected to the same network.

NAME

fs – file system format of system volume

SYNOPSIS

```
#include <sys/filsys.h>
#include <sys/types.h>
#include <sys/param.h>
```

DESCRIPTION

Every file system storage volume has a common format for certain vital information. Every such volume is divided into a certain number of 512 byte long sectors. Sector 0 is unused and is available to contain a bootstrap program or other information.

Sector 1 is the *super-block*. The format of a super-block is:

```
/*
 * Structure of the super-block
 */
struct  filsys {
        unsigned short s_isize; /* size in blocks of i-list */
        daddr_t s_fsize;        /* size in blocks of entire volume */
        short   s_nfree;        /* number of addresses in s_free */
        daddr_t s_free[NICFREE];/* free block list */
        short   s_ninode;       /* number of i-nodes in s_inode */
        ino_t   s_inode[NICINOD];/* free i-node list */
        char    s_flock;        /* lock during free list manipulation */
        char    s_ilock;        /* lock during i-list manipulation */
        char    s_fmod;         /* super block modified flag */
        char    s_ronly;        /* mounted read-only flag */
        time_t  s_time;         /* last super block update */
        short   s_dinfo[4];     /* device information */
        daddr_t s_tfree;        /* total free blocks*/
        ino_t   s_tinode;       /* total free inodes */
        char    s_fname[6];     /* file system name */
        char    s_fpack[6];     /* file system pack name */
        long    s_fill[15];     /* ADJUST to make sizeof filsys be 512 */
        long    s_magic;        /* magic number to indicate new file system */
        long    s_type;         /* type of file system */
};

#define FsMAGIC 0xfd187e20

#define Fs1b    1               /* 512 byte blocks */
#define Fs2b    2               /* 1024 byte blocks */
```

S_type indicates the file system type. Currently, two types of file systems are supported: the original 512-byte oriented and the new improved 1024-byte oriented. *S_magic* is used to distinguish the original 512-byte oriented file systems from the newer file systems. If this field is not equal to the magic number, *FsMAGIC*, the type is assumed to be *Fs1b*, otherwise the *s_type* field is used. In the following description, a block is then determined by the type. For the original 512-byte oriented file system, a block is 512 bytes. For the 1024-byte oriented file system, a block is 1024 bytes or two sectors. The operating system takes care of all conversions from logical block numbers to physical sector numbers.

S_isize is the address of the first data block after the i-list; the i-list starts just after the super-block, namely in block 2; thus the i-list is *s_isize*−2 blocks long. *S_fsize* is the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block numbers; if an "impossible" block number is allocated from the free list or is freed, a diagnostic is written on the on-line console.

Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *s_free* array contains, in *s_free*[1], ..., *s_free*[*s_nfree*−1], up to 49 numbers of free blocks. *S_free*[0] is the block number of the head of a chain of blocks constituting the free list. The first long in each free-chain block is the number (up to 50) of free-block numbers listed in the next 50 longs of this chain member. The first of these 50 blocks is the link to the next member of the chain. To allocate a block: decrement *s_nfree*, and the new block is *s_free*[*s_nfree*]. If the new block number is 0, there are no blocks left, so give an error. If *s_nfree* became 0, read in the block named by the new block number, replace *s_nfree* by its first word, and copy the block numbers in the next 50 longs into the *s_free* array. To free a block, check if *s_nfree* is 50; if so, copy *s_nfree* and the *s_free* array into it, write it out, and set *s_nfree* to 0. In any event set *s_free*[*s_nfree*] to the freed block's number and increment *s_nfree*.

S_tfree is the total free blocks available in the file system.

S_ninode is the number of free i-numbers in the *s_inode* array. To allocate an i-node: if *s_ninode* is greater than 0, decrement it and return *s_inode*[*s_ninode*]. If it was 0, read the i-list and place the numbers of all free inodes (up to 100) into the *s_inode* array, then try again. To free an i-node, provided *s_ninode* is less than 100, place its number into *s_inode*[*s_ninode*] and increment *s_ninode*. If *s_ninode* is already 100, do not bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

S_tinode is the total free inodes available in the file system.

S_flock and *s_ilock* are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of *s_fmod* on disk is likewise immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information.

S_ronly is a read-only flag to indicate write-protection.

S_time is the last time the super-block of the file system was changed, and is the number of seconds that have elapsed since 00:00 Jan. 1, 1970 (GMT). During a reboot, the *s_time* of the super-block for the root file system is used to set the system's idea of the time.

S_fname is the name of the file system and *s_fpack* is the name of the pack.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. Also, i-nodes are 64 bytes long. I-node 1 is reserved for future use. I-node 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. For the format of an inode and its flags, see *inode*(5).

FILES

/usr/include/sys/filsys.h
/usr/include/sys/stat.h

SEE ALSO

fsck(8), fsdb(8), mkfs(8), inode(5).

NAME

fspec – format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on the UNIX System with non-standard tabs, (i.e., tabs which are not set at every eighth column). Such files must generally be converted to a standard format, frequently by replacing all tabs with the appropriate number of spaces, before they can be processed by UNIX System commands. A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

A format specification consists of a sequence of parameters separated by blanks and surrounded by the brackets <: and :>. Each parameter consists of a keyletter, possibly followed immediately by a value. The following parameters are recognized:

t*tabs* The **t** parameter specifies the tab settings for the file. The value of *tabs* must be one of the following:

1. a list of column numbers separated by commas, indicating tabs set at the specified columns;
2. a – followed immediately by an integer *n*, indicating tabs at intervals of *n* columns;
3. a – followed by the name of a "canned" tab specification.

Standard tabs are specified by **t–8**, or equivalently, **t1,9,17,25**,etc. The canned tabs which are recognized are defined by the *tabs*(1) command.

s*size* The **s** parameter specifies a maximum line size. The value of *size* must be an integer. Size checking is performed after tabs have been expanded, but before the margin is prepended.

m*margin*
The **m** parameter specifies a number of spaces to be prepended to each line. The value of *margin* must be an integer.

d The **d** parameter takes no value. Its presence indicates that the line containing the format specification is to be deleted from the converted file.

e The **e** parameter takes no value. Its presence indicates that the current format is to prevail only until another format specification is encountered in the file.

Default values, which are assumed for parameters not supplied, are **t–8** and **m0**. If the **s** parameter is not specified, no size checking is performed. If the first line of a file does not contain a format specification, the above defaults are assumed for the entire file. The following is an example of a line containing a format specification:

```
* <:t5,10,15 s72:> *
```

If a format specification can be disguised as a comment, it is not necessary to code the **d** parameter.

Several UNIX System commands correctly interpret the format specification for a file. Among them is *gath* (see *send*(1C)) which may be used to convert files to a standard format acceptable to other UNIX System commands.

SEE ALSO

ed(1), send(1C), tabs(1).

NAME

gettydefs – speed and terminal settings used by getty

DESCRIPTION

The **/etc/gettydefs** file contains information used by *getty*(8) to set up the speed and terminal settings for a line. It supplies information on what the *login* prompt should look like. It also supplies the speed to try next if the user indicates the current speed is not correct by typing a <*break*> character.

Each entry in **/etc/gettydefs** has the following format:

label# initial-flags # final-flags # login-prompt #next-label

Each entry is followed by a blank line. Lines that begin with # are ignored and may be used to comment the file. The various fields can contain quoted characters of the form **\b**, **\n**, **\c**, etc., as well as **\***nnn*, where *nnn* is the octal value of the desired character. The various fields are:

label	This is the string against which *getty* tries to match its second argument. It is often the speed, such as 1200, at which the terminal is supposed to run, but it needn't be (see below).
initial-flags	These flags are the initial *ioctl*(2) settings to which the terminal is to be set if a terminal type is not specified to *getty*. *Getty* understands the symbolic names specified in **/usr/include/sys/termio.h** (see *termio*(4)). Normally only the speed flag is required in the *initial-flags*. *Getty* automatically sets the terminal to raw input mode and takes care of most of the other flags. The *initial-flag* settings remain in effect until *getty* executes *login*(1).
final-flags	These flags take the same values as the *initial-flags* and are set just prior to *getty* executes *login*. The speed flag is again required. The composite flag **SANE** takes care of most of the other flags that need to be set so that the processor and terminal are communicating in a rational fashion. The other two commonly specified *final-flags* are **TAB3**, so that tabs are sent to the terminal as spaces, and **HUPCL** so that the line is hung up on the final close.
login-prompt	This entire field is printed as the *login-prompt*. Unlike the above fields where white space is ignored (a space, tab or new-line), they are included in the *login-prompt* field.
next-label	This indicates the next *label* of the entry in the table that *getty* should use if the user types a <*break*> or the input cannot be read. Usually, a series of speeds are linked together in this fashion, into a closed set. For instance, 2400 linked to 1200, which in turn is linked to 300, which finally is linked to 2400.

If *getty* is called without a second argument, then the first entry of **/etc/gettydefs** is used, thus making the first entry of **/etc/gettydefs** the default entry. It is also used if *getty* can't find the specified *label*. If **/etc/gettydefs** itself is missing, there is one entry built into the command which will bring up a terminal at 300 baud.

It is strongly recommended that after making or modifying **/etc/gettydefs**, it be run through *getty* with the check option to be sure there are no errors.

FILES
/etc/gettydefs

SEE ALSO
getty(8), termio(4)
login(1), ioctl(2).

NAME

group – group file

DESCRIPTION

Group contains for each group the following information:

group name
encrypted password
numerical group ID
a comma separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; Each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

FILES

/etc/group

SEE ALSO

newgrp(1), crypt(3C), passwd(1), passwd(5)

NAME

inittab – script for the init process

DESCRIPTION

The *inittab* file supplies the script to *init*'s role as a general process dispatcher. The process that constitutes the majority of *init*'s process dispatching activities is the line process */etc/getty* that initiates individual terminal lines. Other processes typically dispatched by *init* are daemons and the shell.

The *inittab* file is composed of entries that are position dependent and have the following format:

```
id:rstate:action:process
```

Each entry is delimited by a newline, however, a backslash (\) preceding a newline indicates a continuation of the entry. Up to 512 characters per entry are permitted. Comments may be inserted in the *process* field using the *sh*(1) convention for comments. Comments for lines that spawn *getty*s are displayed by the *who*(1) command. It is expected that they will contain some information about the line such as the location. There are no limits (other than maximum entry size) imposed on the number of entries within the *inittab* file. The entry fields are:

id This is one to four characters used to uniquely identify an entry.

rstate This defines the *run-level* in which this entry is to be processed. *Run-levels* effectively correspond to a configuration of processes in the system. That is, each process spawned by *init* is assigned a *run-level* or *run-levels* in which it is allowed to exist. The *run-levels* are represented by a number ranging from 0 through 6. As an example, if the system is in *run-level* 1, only those entries having a 1 in the *rstate* field will be processed. When *init* is requested to change *run-levels*, all processes which do not have an entry in the *rstate* field for the target *run-level* will be sent the warning signal (**SIGTERM**) and allowed a 20 second grace period before being forcibly terminated by a kill signal (**SIGKILL**). The *rstate* field can define multiple *run-levels* for a process by selecting more than one *run-level* in any combination from **0**–**6**. If no *run-level* is specified, then *action* will be taken on this *process* for all *run-levels* **0**–**6**. There are three other values, **a**, **b** and **c**, which can appear in the *rstate* field, even though they are not true *run-levels*. Entries which have these characters in the *rstate* field are processed only when the *telinit* (see *init*(8)) process requests them to be run (regardless of the current *run-level* of the system). They differ from *run-levels* in that the system is only in these states for as long as it takes to execute all the entries associated with the states. A process started by an **a**, **b** or **c** command is not killed when *init* changes levels. They are only killed if their line in **/etc/inittab** is marked **off** in the *action* field, their line is deleted entirely from **/etc/inittab**, or *init* goes into the *SINGLE USER* state.

action Key words in this field tell *init* how to treat the process specified in the *process* field. The actions recognized by *init* are as follows:

respawn If the process does not exist then start the process, do not wait for its termination (continue scanning the *inittab* file), and when it dies restart the process. If the process currently exists then do nothing and continue scanning the *inittab* file.

wait Upon *init*'s entering the *run-level* that matches the entry's *rstate*, start the process and wait for its termination. All subsequent reads of the *inittab* file while *init* is in the same *run-level* will cause *init* to ignore this entry.

once Upon *init*'s entering a *run-level* that matches the entry's *rstate*, start the process, do not wait for its termination and when it dies, do not restart the process. If upon entering a new *run-level*, where the process is still running from a previous *run-level* change, the program will not be restarted.

boot The entry is to be processed only at *init*'s boot-time read of the *inittab* file. *Init* is to start the process, not wait for its termination, and when it dies, not restart the process. In order for this instruction to be meaningful, the *rstate* should be the default or it must match *init*'s *run-level* at boot time. This action is useful for an initialization function following a hardware reboot of the system.

bootwait The entry is to be processed only at *init*'s boot-time read of the *inittab* file. *Init* is to start the process, wait for its termination and, when it dies, not restart the process.

powerfail Execute the process associated with this entry only when *init* receives a power fail signal (SIGPWR see *signal*(2)).

powerwait Execute the process associated with this entry only when *init* receives a power fail signal (SIGPWR) and wait until it terminates before continuing any processing of *inittab*.

off If the process associated with this entry is currently running, send the warning signal (SIGTERM) and wait 20 seconds before forcibly terminating the process via the kill signal (SIGKILL). If the process is nonexistent, ignore the entry.

ondemand This instruction is really a synonym for the **respawn** action. It is functionally identical to **respawn** but is given a different keyword in order to divorce its association with *run-levels*. This is used only with the **a**, **b** or **c** values described in the *rstate* field.

initdefault An entry with this *action* is only scanned when *init* initially invoked. *Init* uses this entry, if it exists, to determine which *run-level* to enter initially. It does this by taking the highest *run-level* specified in the **rstate** field and using that as its initial state. If the *rstate* field is empty, this is interpreted as **0123456** and so *init* will enter *run-level* **6**. Also, the **initdefault** entry can use **s** to specify that *init* start in the *SINGLE USER* state. Additionally, if *init* doesn't find an **initdefault** entry in **/etc/inittab**, then it will request an initial *run-level* from the user at reboot time.

sysinit Entries of this type are executed before *init* tries to access the console. It is expected that this entry will be only used to initialize devices on which *init* might try to ask the *run-level* question. These entries are executed and waited for before continuing.

process This is a *sh* command to be executed. The entire **process** field is prefixed with *exec* and passed to a forked *sh* as **sh −c 'exec** *command'*. For this reason, any legal *sh* syntax can appear in the *process* field. Comments can be inserted with the *# comment* syntax.

FILES

/etc/inittab

SEE ALSO

getty(8), init(8)
sh(1), who(1), exec(2), open(2), signal(2).

NAME

inode − format of an inode

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ino.h>
```

DESCRIPTION

An i-node for a plain file or directory in a file system has the following structure defined by <sys/ino.h>.

```
/*
 * Inode structure as it appears on
 * a disk block.
 */
struct dinode
{
        unsigned short di_mode;         /* mode and type of file */
        short   di_nlink;       /* number of links to file */
        short   di_uid;         /* owner's user id */
        short   di_gid;         /* owner's group id */
        off_t   di_size;        /* number of bytes in file */
        char    di_addr[40];    /* disk block addresses */
        time_t  di_atime;       /* time last accessed */
        time_t  di_mtime;       /* time last modified */
        time_t  di_ctime;       /* time created */
};

/*
 * the 40 address bytes:
 *      39 used; 13 addresses
 *      of 3 bytes each.
 */
```

For the meaning of the defined types *off_t* and *time_t* see *types*(7).

FILES

/usr/include/sys/ino.h

SEE ALSO

stat(2), fs(5), types(7).

NAME

issue − issue identification file

DESCRIPTION

The file **/etc/issue** contains the *issue* or project identification to be printed as a login prompt. This is an ASCII file which is read by program *getty* and then written to any terminal spawned or respawned from the *lines* file.

FILES

/etc/issue

SEE ALSO

login(1).

NAME
keycap – keyboard capability data base

SYNOPSIS
/etc/keycap

DESCRIPTION
Keycap parametrizes keyboard input from different terminals, much as *termcap* parametrizes output. For example, the following lines in */etc/keycap*

```
        pv|pcsdsg|vt100/52|pcs vt100:
          .
          .
          .
        :\ED=#8c:      /left
          .
          .
          .
```

describe the $\boxed{\leftarrow}$ key on a VT100 (VT52 mode). Pressing $\boxed{\leftarrow}$ on the keyboard sends two characters (i.e. <ESC><D>). *Keycap* decodes the input stream from the keyboard, and delivers one byte (i.e. 0x8c).

See *termcap(5)* for the first entry for each specific terminal in the keyboard capability data base. Each further line indicates a byte-sequence translation. Each translation is enclosed by colons ':'. The source byte-sequence and the target are separated by an equal sign '='.

The following characters should be escaped by a backslash '\':

```
        =:^\#
```

The usual C-string convention applies to octal-constants and the following sequences:

```
        \b  \f  \n  \r  \t
```

Hexadecimal constants start with the character '#'.

A control character is started by a '^' (i.e. ^A means <CTRL A> = \001).

Backslash followed by an uppercase E means <ESCAPE> (i.e. \033).

FILES
/etc/keycap   file containing keyboard descriptions

SEE ALSO
termcap(3)

AUTHOR
Dittmar Krall

NAME

mnttab – mounted file system table

SYNOPSIS

```
struct mnttab {
        char    mt_dev[10];
        char    mt_filsys[10];
        short   mt_ro_flg;
        time_t  mt_time;
};
```

DESCRIPTION

Mnttab resides in directory /etc and contains a table of devices mounted by the *mount*(8) command.

Each entry is 26 bytes in length; the first 10 bytes are the null-padded name of the place where the *special file* is mounted; the next 10 bytes represent the null-padded root name of the mounted special file; the remaining 6 bytes contain the mounted *special file*'s read/write permissions and the date on which it was mounted.

The maximum number of entries in *mnttab* is based on the system parameter NMOUNT located in /usr/sys/conf.h, which defines the number of allowable mounted special files.

SEE ALSO

mount(8).

NAME

news – USENET network news article, utility files

DESCRIPTION

There are two formats of news articles: A and B. A format is the only format that version 1 netnews systems can read or write. Systems running the version 2 netnews can read either format and there are provisions for the version 2 netnews to write in A format. A format looks like this:

Aarticle-ID
newsgroups
path
date
title
Body of article

Only version 2 netnews systems can read and write B format. B format contains two extra pieces of information: receival date and expiration date. The basic structure of a B format file consists of a series of headers and then the body. A header field is defined as a line with a capital letter in the 1st column and a colon somewhere on the line. Unrecognized header fields are ignored. News is stored in the same format transmitted, see "Standard for the Interchange of USENET Messages" for a full description. The following fields are among those recognized:

Header	Information
From:	*user@host.domain*[*.domain* ...] *(Full Name)*
Newsgroups:	*Newsgroups*
Message-ID:	*<Unique Identifier>*
Subject:	*descriptive title*
Date:	*Date Posted*
Date-Received:	*Date received on local machine*
Expires:	*Expiration Date*
Reply-To:	*Address for mail replies*
References:	*Article ID of article this is*
Control:	*Text of a control message*

Here is an example of an article:

```
Relay-Version: B 2.10      2/13/83 cbosgd.UUCP
Posting-Version: B 2.10    2/13/83 eagle.UUCP
Path: cbosgd!mhuxj!mhuxt!eagle!jerry
From: jerry@eagle.uucp (Jerry Schwarz)
Newsgroups: net.general
Subject: Usenet Etiquette -- Please Read
Message-ID: <642@eagle.UUCP>
Date: Friday, 19-Nov-82 16:14:55 EST
Followup-To: net.news
Expires: Saturday, 1-Jan-83 00:00:00 EST
```

```
Date-Received: Friday, 19-Nov-82 16:59:30 EST
Organization: Bell Labs, Murray Hill
```

The body of the article comes here, after a blank line.

The files mentioned next all reside in /usr/lib/news. A *sys* file line has four fields, each seperated by colons:

system-name:subscriptions:flags:transmission command

Of these fields, on the *system-name* and *subscriptions* need to be present.

The *system name* is the name of the system being sent to. The *subscriptions* is the list of newsgroups to be transmitted to the system. The *flags* are a set of letters describing how the article should be transmitted. The default is B. Valid flags include A (send in A format), B (send in B format), N (use ihave/sendme protocol), U (use uux -c and the name of the stored article in a %s string).

The *transmission command* is executed by the shell with the article to be transmitted as the standard input. The default is **uux – –z –r** *sysname*!**rnews**. Some examples:

```
xyz:net.all
oldsys:net.all,fa.all,to.oldsys:A
berksys:net.all,ucb.all::/usr/lib/news/sendnews —b berksys\:rnews
arpasys:net.all,arpa.all::/usr/lib/news/sendnews —a rnews@arpasys
old2:net.all,fa.all:A:/usr/lib/sendnews —o old2\:rnews
user:fa.sf-lovers::mail user
```

Somewhere in a *sys* file, there must be a line for the host system. This line has no *flags* or *commands*. A # as the first character in a line denotes a comment.

The history, active, and ngfile files have one line per item.

SEE ALSO

inews(1), postnews(1), sendnews(8), uurec(8), readnews(1)

NAME

newsrc − information file for readnews(1) and checknews(1)

DESCRIPTION

The *.newsrc* file contains the list of previously read articles and an optional options line for *readnews(1)* and *checknews(1)*. Each newsgroup that articles have been read from has a line of the form:

newsgroup: range

The *range* is a list of the articles read. It is basically a list of no.'s separated by commas with sequential no.'s collapsed with hyphens. For instance:

general: 1-78,80,85-90
fa.info-cpm: 1-7
net.news: 1
fa.info-vax! 1-5

If the : is replaced with an ! (as in info-vax above) the newsgroup is not subscribed to and will not be shown to the user.

An options line starts with the word **options** (left-justified). Then there are the list of options just as they would be on the command line. For instance:

options −n all !fa.sf-lovers !fa.human-nets −r
options −c −r

A string of lines beginning with a space or tab after the initial options line will be considered continuation lines.

FILES

~/.newsrc	options and list of previously read articles

SEE ALSO

readnews(1), checknews(1)

NAME

passwd – password file

DESCRIPTION

Passwd contains for each user the following information:

name (login name, contains no upper case)
encrypted password
numerical user ID
numerical group ID
Real name of user, office, etc
initial working directory
program to use as Shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, the Shell itself is used.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user ID's to names.

EXAMPLE

```
jones:ztlFqtcoDzINU:14:2:Ed Jones,Big Company,787375:/user/jones:/bin/sh
```

FILES

/etc/passwd

SEE ALSO

getpwent(3C), login(1), crypt(3C), passwd(1), group(5)

[illegible]

NAME

plot − graphics interface

DESCRIPTION

Files of this format are produced by routines described in *plot*(3G), and are interpreted for various devices by commands described in *plot*(1G). A graphics file is a stream of plotting instructions. Each instruction consists of an ASCII letter usually followed by bytes of binary information. The instructions are executed in order. A point is designated by four bytes representing the x and y values; each value is a signed integer. The last designated point in an **l**, **m**, **n**, or **p** instruction becomes the 'current point' for the next instruction.

Each of the following descriptions begins with the name of the corresponding routine in *plot*(3G).

m move: The next four bytes give a new current point.

n cont: Draw a line from the current point to the point given by the next four bytes. See *plot*(1G).

p point: Plot the point given by the next four bytes.

l line: Draw a line from the point given by the next four bytes to the point given by the following four bytes.

t label: Place the following ASCII string so that its first character falls on the current point. The string is terminated by a newline.

a arc: The first four bytes give the center, the next four give the starting point, and the last four give the end point of a circular arc. The least significant coordinate of the end point is used only to determine the quadrant. The arc is drawn counter-clockwise.

c circle: The first four bytes give the center of the circle, the next two the radius.

e erase: Start another frame of output.

f linemod: Take the following string, up to a newline, as the style for drawing further lines. The styles are 'dotted,' 'solid,' 'longdashed,' 'shortdashed,' and 'dotdashed.' Effective only in *plot 4014* and *plot ver*.

s space: The next four bytes give the lower left corner of the plotting area; the following four give the upper right corner. The plot will be magnified or reduced to fit the device as closely as possible.

Space settings that exactly fill the plotting area with unity scaling appear below for devices supported by the filters of *plot*(1G). The upper limit is just outside the plotting area.

```
4014      space(0, 0, 3120, 3120);
lbp (LBP-10)
          space(0, 0, 1696, 2500);
V80 (Versatec)
          space(0, 0, 1536, 1536);
300, 300s
          space(0, 0, 4096, 4096);
450       space(0, 0, 4096, 4096);
```

SEE ALSO
plot(1G), plot(3G), graph(1G)

NAME

profile – setting up an environment at login time

DESCRIPTION

If your login directory contains a file named **.profile**, that file will be executed (via the shell's **exec .profile**) before your session begins; **.profiles** are handy for setting exported environment variables and terminal modes. If the file **/etc/profile** exists, it will be executed for every user before the **.profile**. The following example is typical (except for the comments):

```
# Make some environment variables global
export MAIL PATH TERM
# Set file creation mask
umask 22
# Tell me when new mail comes in
MAIL=/usr/mail/myname
# Add my /bin directory to the shell search sequence
PATH=$PATH:$HOME/bin
```

FILES

$HOME/.profile
/etc/profile

SEE ALSO

env(1), login(1), mail(1), sh(1), stty(1), su(1), environ(7), term(7).

NAME

/etc/pwmap, /etc/groupmap – table of user and group id, mappings for the Newcastle Connection at this system.

DESCRIPTION

/etc/pwmap and */etc/groupmap* contain the tables used by the spawner at this system to determine the user and group ids of servers run on this system on behalf of users of a remote system. The formats of the two files are identical, and consist of a list of system entries, one for each remote system for which one or more users has been authorised. Each system entry consists of a header, and a sequence of fixed-length records for each mapping of a remote id. Each record consists of three 16-bit integers: the first contains flag bits unused in Release 1.0, and the next two contain the remote numeric id and the local numeric id to which it is mapped, respectively.

The header for each system consists of a 16-bit integer giving the number of remote user entries following, a 16-bit length referring to the string name which follows, and a variable-length string which is the path-name of the remote system relative to this system's root. The length field includes the null byte terminating the string.

SEE ALSO

unite (8N), mksys(8N), rmsys(8N).

FILES

/etc/pwmap, /etc/groupmap

NAME

sccsfile – format of SCCS file

DESCRIPTION

An SCCS file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains login names and/or numerical group IDs of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the ASCII SOH (start of heading) character (octal 001). This character is hereafter referred to as *the control character* and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form **DDDDD** represent a five digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

```
@hDDDDD
```

The value of the checksum is the sum of all characters, except those of the first line. The @h provides a *magic number* of (octal) 064001.

Delta table

The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID>  yr/mo/da hr:mi:se  <pgmr>  DDDDD  DDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
  .
  .
  .
@c <comments> ...
  .
  .
  .
@e
```

The first line (@s) contains the number of lines inserted/deleted/unchanged respectively. The second line (@d) contains the type of the delta (currently, normal: **D**, and removed: **R**), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The @i, @x, and @g lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The @m lines (optional) each contain one **MR** number associated with the delta; the @c lines contain comments associated with the delta.

The @e line ends the delta table entry.

User names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines @u and @U. An empty list allows anyone to make a delta.

Flags

Keywords used internally (see *admin*(1) for more information on their use). Each flag line takes the form:

```
@f <flag>     <optional text>
```

The following flags are defined:

```
@f t   <type of program>
@f v   <program name>
@f i
@f b
@f m   <module name>
@f f   <floor>
@f c   <ceiling>
@f d   <default-sid>
@f n
@f j
@f l   <lock-releases>
@f q   <user defined>
```

The **t** flag defines the replacement for the **%Y%** identification keyword. The **v** flag controls prompting for **MR** numbers in addition to comments; if the optional text is present it defines an **MR** number validity checking program. The **i** flag controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the **b** flag is present the **−b** keyletter may be used on the *get* command to cause a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the **%M%** identification keyword. The **f** flag defines the "floor" release; the release below which no deltas may be added.

The **c** flag defines the "ceiling" release; the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a *get* command. The **n** flag causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes *get* to allow concurrent edits of the same base SID. The **l** flag defines a *list* of releases that are *locked* against editing (*get*(1) with the **−e** keyletter). The **q** flag defines the replacement for the **%Q%** identification keyword.

Comments

Arbitrary text surrounded by the bracketing lines @t and @T. The comments section typically will contain a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines don't begin with the control character, control lines do. There are three kinds of control lines: *insert*, *delete*, and *end*, represented by:

```
@I DDDDD
@D DDDDD
@E DDDDD
```

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1), delta(1), get(1), prs(1).
Source Code Control System User's Guide by L. E. Bonanni and C. A. Salemi.

NAME

C stack frame layout

DESCRIPTION

This is a typical procedure call compiled from the statement

read(fildes, buf, 1024);

```
move.w   #400,-(a7)         push the constant #400  (2 bytes)
pea      _buf               push address of buf     (4 bytes)
move.w   _fildes,-(a7)      push fildes             (2 bytes)
jsr      _read              call _read
addq.w   #8,a7              pop 8 = 2+4+2 bytes
```

Parameters are pushed on the stack, the procedure is called, and after return the parameters are popped again. A typical procedure entry and exit looks like this:

```
_read:  link a6,#-30              save old a6, reserve 30 bytes
        movem.l d6/d7/a5,-24(a6)  save register variables
        ...
        move.l  result,d0         result transferred via d0
        movem.l -24(a6),d6/d7/a5  restore register variables
        unlk    a6                restore old a6, release frame
        rts                       return to caller
```

The link instruction makes room on the stack for

a) the register saving area. This is a constant 24 byte long area for a maximum of 6 register variables, 3 each for data and address registers.

b) a word used for storing the current line number, if the -L option of cc is used.

c) the local variables.

The first movem saves three registers into the register area. Exactly the same three registers are restored with the second movem. Only those registers among a3-a5,d5-d7 are saved, that are modified in the procedure. If none of these registers is modified, than both movem instructions are suppressed.

A function result is returned in D0 (F0 for certain floating point formats). The unlk returns the frame, and the rts returns to the caller.
The stack frame format follows from the given code:

```
          |                    |                high address
          |                    |
          |   Parameters       |  <- 8(A6)
          +--------------------+
          | return address     |
          | (written by jsr)   |  <- 4(A6)
          +--------------------+
          | old  A6            |
          | (saved by link)    |  <- (A6)
          +--------------------+
          |                    |
          |  Register          |
          |  Saving            |
          |  Area              |
          |                    |
          |  (24 = 6 * 4)      |
          |                    |
          |         .          |  <- -24(A6)
          +--------------------+
          | cur. lineno        |  <- -26(A6)
          +--------------------+
          | local variables    |                low address
          |                    |
```

NAME

termcap — terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminals, used, *e.g.*, by *vi*(1) and *curses*(3). Terminals are described in *termcap* by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in *termcap*.

Entries in *termcap* consist of a number of ':' separated fields. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name is always 2 characters long and is used by older systems which store the terminal type in a 16 bit word in a systemwide data base. The second name given is the most common abbreviation for the terminal, and the last name given should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may well contain blanks for readability.

CAPABILITIES

(P) indicates padding may be specified
(P*) indicates that padding may be based on no. lines affected

Name	Type	Pad?	Description
ae	str	(P)	End alternate character set
al	str	(P*)	Add new blank line
am	bool		Terminal has automatic margins
as	str	(P)	Start alternate character set
bc	str		Backspace if not ^H
bs	bool		Terminal can backspace with ^H
bt	str	(P)	Back tab
bw	bool		Backspace wraps from column 0 to last column
CC	str		Command character in prototype if terminal settable
cd	str	(P*)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(P)	Like cm but horizontal motion only, line stays same
cl	str	(P*)	Clear screen
cm	str	(P)	Cursor motion
co	num		Number of columns in a line
cr	str	(P*)	Carriage return, (default ^M)
cs	str	(P)	Change scrolling region (vt100), like cm
cv	str	(P)	Like ch but vertical only.
da	bool		Display may be retained above
dB	num		Number of millisec of bs delay needed
db	bool		Display may be retained below
dC	num		Number of millisec of cr delay needed
dc	str	(P*)	Delete character
dF	num		Number of millisec of ff delay needed
dl	str	(P*)	Delete line
dm	str		Delete mode (enter)
dN	num		Number of millisec of nl delay needed

do	str		Down one line
dT	num		Number of millisec of tab delay needed
ed	str		End delete mode
ei	str		End insert mode; give :ei=: if **ic**
eo	str		Can erase overstrikes with a blank
ff	str	(P*)	Hardcopy terminal page eject (default ^L)
hc	bool		Hardcopy terminal
hd	str		Half-line down (forward 1/2 linefeed)
ho	str		Home cursor (if no **cm**)
hu	str		Half-line up (reverse 1/2 linefeed)
hz	str		Hazeltine; can't print ~'s
ic	str	(P)	Insert character
if	str		Name of file containing **is**
im	bool		Insert mode (enter); give :im=: if **ic**
in	bool		Insert mode distinguishes nulls on display
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string
k0-k9	str		Sent by other function keys 0-9
kb	str		Sent by backspace key
kd	str		Sent by terminal down arrow key
ke	str		Out of keypad transmit mode
kh	str		Sent by home key
kl	str		Sent by terminal left arrow key
kn	num		Number of other keys
ko	str		Termcap entries for other non-function keys
kr	str		Sent by terminal right arrow key
ks	str		Put terminal in keypad transmit mode
ku	str		Sent by terminal up arrow key
l0-l9	str		Labels on other function keys
li	num		Number of lines on screen or page
ll	str		Last line, first column (if no **cm**)
ma	str		Arrow key map, used by vi version 2 only
mi	bool		Safe to move while in insert mode
ml	str		Memory lock on above cursor.
ms	bool		Safe to move while in standout and underline mode
mu	str		Memory unlock (turn off memory lock).
nc	bool		No correctly working carriage return (DM2500,H2000)
nd	str		Non-destructive space (cursor right)
nl	str	(P*)	Newline character (default **\n**)
ns	bool		Terminal is a CRT but doesn't scroll.
os	bool		Terminal overstrikes
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with **is**)
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)
ta	str	(P)	Tab (other than ^I or with padding)
tc	str		Entry of similar terminal - must be last
te	str		String to end programs that use **cm**
ti	str		String to begin programs that use **cm**

uc	str	Underscore one char and move past it
ue	str	End underscore mode
ug	num	Number of blank chars left by us or ue
ul	bool	Terminal underlines even though it doesn't overstrike
up	str	Upline (cursor up)
us	str	Start underscore mode
vb	str	Visible bell (may not move cursor)
ve	str	Sequence to end open/visual mode
vs	str	Sequence to start open/visual mode
xb	bool	Beehive (f1=escape, f2=ctrl C)
xn	bool	A newline is ignored after a wrap (Concept)
xr	bool	Return acts like **ce** \r \n (Delta Data)
xs	bool	Standout not erased by writing over it (HP 264?)
xt	bool	Tabs are destructive, magic so char (Teleray 1061)

A Sample Entry

The following entry, which describes the Concept−100, is among the more complex entries in the *termcap* file as of this writing. (This particular concept entry is outdated, and is used as an example only.)

```
c1|c100|concept100:is=\EU\Ef\E7\E5\E8\El\ENH\EK\E\200\Eo&\200:\
      :al=3*\E^R:am:bs:cd=16*\E^C:ce=16\E^S:cl=2*^L:cm=\Ea%+ %+
      :co#80:\:dc=16\E^A:dl=3*\E^B:ei=\E\200:eo:im=\E^P:in
      :ip=16*:li#24:mi:nd=\E=:\
      :se=\Ed\Ee:so=\ED\EE:ta=8\t:ul:up=\E;:vb=\Ek\EK:xn:
```

Entries may continue onto multiple lines by giving a \ as the last character of a line, and that empty fields may be included for readability (here between the last field on a line and the first field on the next). Capabilities in *termcap* are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have two letter codes. For instance, the fact that the Concept has automatic margins (i.e. an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **co** which indicates the number of columns the terminal has gives the value '80' for the Concept.

Finally, string valued capabilities, such as **ce** (clear to end of line sequence) are given by the two character code, an '=', and then a string ending at the next following ':'. A delay in milliseconds may appear after the '=' in such a capability, and padding characters are supplied by the editor after the remainder of the string is sent to provide this delay. The delay can be either a integer, e.g. '20', or an integer followed by an '*', i.e. '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' to specify a delay per unit to tenths of milliseconds.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. A **\E** maps to an ESCAPE character, **^x** maps to a control-x for any appropriate x, and the sequences **\n \r \t \b \f** give a newline, return, tab, backspace and formfeed. Finally, characters may be given as three octal digits after **a **, and the characters **^** and **** may be given as **\^** and \. If it is necessary to place a : in a capability it must be escaped in octal as **\072**. If it is necessary to place a null character in a string capability it must be encoded as **\200**. The routines which deal with *termcap* use C strings, and strip the high bits of the output very late so that a **\200** comes out as a **\000** would.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *termcap* and to build up a description gradually, using partial descriptions with *ex* to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the *termcap* file to describe it or bugs in *ex*. To easily test a new terminal description you can set the environment variable TERMCAP to a pathname of a file containing the description you are working on and the editor will look there rather than in */etc/termcap*. TERMCAP can also be set to the termcap entry itself to avoid reading the file when starting up the editor.

Basic capabilities

The number of columns on each line for the terminal is given by the **co** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **li** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, then this is given by the **cl** string capability. If the terminal can backspace, then it should have the **bs** capability, unless a backspace is accomplished by a character other than ^H in which case you should give this character as the **bc** string capability. If it overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability.

A very important point here is that the local cursor motions encoded in *termcap* are undefined at the left and top edges of a CRT terminal. The editor will never attempt to backspace around the left edge, nor will it attempt to go up locally off the top. The editor assumes that feeding off the bottom of the screen will cause the screen to scroll up, and the **am** capability tells whether the cursor sticks at the right edge of the screen. If the terminal has switch selectable automatic margins, the *termcap* file usually assumes that this is on, i.e. **am**.

These capabilities suffice to describe hardcopy and glass-tty terminals. Thus the model 33 teletype is described as

```
        t3|33|tty33:co#72:os
```

while the Lear Siegler ADM-3 is described as

```
cl|adm3|3|lsi adm3:am:bs:cl=^Z:li#24:co#80
```

Cursor addressing

Cursor addressing in the terminal is described by a **cm** string capability, with *printf*(3S) like escapes **%x** in it. These substitute to encodings of the current line or column position, while other characters are passed through unchanged. If the **cm** string is thought of as being a function, then its arguments are the line and then the column to which motion is desired, and the **%** encodings have the following meanings:

%d	as in *printf*, 0 origin
%2	like %2d
%3	like %3d
%.	like %c
%+x	adds *x* to value, then %.
%>xy	if value > x adds y, no output.
%r	reverses order of line and column, no output
%i	increments line/column (for 1 origin)
%%	gives a single %
%n	exclusive or row and column with 0140 (DM2500)
%B	BCD (16*(x/10)) + (x%10), no output.
%D	Reverse coding (x-2*(x%16)), no output. (Delta Data).

Consider the HP2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cm** capability is cm=6\E&%r%2c%2Y. The Microterm ACT-IV needs the current row and column sent preceded by a ^T, with the row and column simply encoded in binary, cm=^T%.%.. Terminals which use %. need to be able to backspace the cursor (**bs** or **bc**), and to move the cursor up one line on the screen (**up** introduced below). This is necessary because it is not always safe to transmit **\t**, **\n** **^D** and **\r**, as the system may change or discard them.

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus cm=\E=%+ %+ .

Cursor motions

If the terminal can move the cursor one position to the right, leaving the character at the current position unchanged, then this sequence should be given as **nd** (non-destructive space). If it can move the cursor up a line on the screen in the same column, this should be given as **up**. If the terminal has no cursor addressing capability, but can home the cursor (to very upper left corner of screen) then this can be given as **ho**; similarly a fast way of getting to the lower left hand corner can be given as **ll**; this may involve going up with **up** from the home position, but the editor will never do this itself (unless **ll** does) because it makes no assumption about the effect of moving up from the home position.

Area clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **ce**. If the terminal can clear from the current position to the end of the display, then this should be given as **cd**. The editor only uses **cd** from the first column of a line.

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **al**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl**; this is done only from the first position on the line to be deleted. If the terminal can scroll the screen backwards, then this can be given as **sb**, but just **al** suffices. If the terminal can retain display memory above then the **da** capability should be given; if display memory can be retained below then **db** should be given. These let the editor understand that deleting a line on the screen may bring non-blank lines up from below or that scrolling back with **sb** may bring down non-blank lines.

Insert/delete character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using *termcap*. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can find out which kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type abc   def using local cursor motions (not spaces) between the abc and the def. Then position the cursor before the abc and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the abc shifts over to the def which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for insert null. If your terminal does something different and unusual then you may have to modify the editor to get it to use the insert mode your terminal defines. We have seen no terminals which have an insert mode not falling into one of these two classes.

The editor can handle both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **im** the sequence to get into insert mode, or give it an empty value if your terminal uses a sequence to insert a blank position. Give as **ei** the sequence to leave insert mode (give this, with an empty value also if you gave **im**). Now give as **ic** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ic**, terminals which send a sequence to open a screen position should give it here. (Insert mode is preferable to the sequence to open a position on the screen if your terminal has both.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g. if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mi** to speed up inserting in this case. Omitting **mi** will affect only speed. Some terminals (notably Datamedia's) must not have **mi** because of the way their insert mode works.

Finally, you can specify delete mode by giving **dm** and **ed** to enter and exit delete mode, and **dc** to delete a single character while in delete mode.

Highlighting, underlining, and visible bells

If your terminal has sequences to enter and exit standout mode these can be given as **so** and **se** respectively. If there are several flavors of standout mode (such as inverse video, blinking, or underlining – half bright is not usually an acceptable standout mode unless the terminal is in inverse video mode constantly) the preferred mode is inverse video by itself. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **ug** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **us** and **ue** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**. (If the underline code does not move the cursor to the right, give the code followed by a nondestructive space.)

Many terminals, such as the HP 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **vb**; it must not move the cursor. If the terminal should be placed in a different mode during open and visual modes of *ex*, this can be given as **vs** and **ve**, sent at the start and end of these modes respectively. These can be used to change, e.g., from a underline to a block cursor and back.

If the terminal needs to be in a special mode when running a program that addresses the cursor, the codes to enter and exit this mode can be given as **ti** and **te**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **ks** and **ke**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kl**, **kr**, **ku**, **kd**, and **kh** respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as **k0**, **k1**, ..., **k9**. If these keys have labels other than the default f0 through f9, the labels can be given as **l0**, **l1**, ..., **l9**. If there are other keys that transmit the same code as the terminal expects for the corresponding function, such as clear screen, the *termcap* 2 letter codes can be given in the **ko** capability, for example, :ko=cl,ll,sf,sb:, which says that the terminal has clear, home down, scroll down, and scroll up keys that transmit the same thing as the cl, ll, sf, and sb entries.

The **ma** entry is also used to indicate arrow keys on terminals which have single character arrow keys. It is obsolete but still in use in version 2 of vi, which must be run on some minicomputers due to memory limitations. This field is redundant with **kl**, **kr**, **ku**, **kd**, and **kh**. It consists of groups of two characters. In each group, the first character is what an arrow key sends, the second character is the corresponding vi command. These commands are **h** for **kl**, **j** for **kd**, **k** for **ku**, **l** for **kr**, and H for **kh**. For example, the mime would be **:ma=^Kj^Zk^Xl:** indicating arrow keys left (^H), down (^K), up (^Z), and right (^X). (There is no home key on the mime.)

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pc**.

If tabs on the terminal require padding, or if the terminal uses a character other than ^I to tab, then this can be given as **ta**.

Hazeltine terminals, which don't allow '~' characters to be printed should indicate **hz**. Datamedia terminals, which echo carriage-return linefeed for carriage return and then ignore a following linefeed should indicate **nc**. Early Concept terminals, which ignore a linefeed immediately after an **am** wrap, should indicate **xn**. If an erase-eol is required to get rid of standout (instead of merely writing on top of it), **xs** should be given. Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt**. Other specific terminal problems may be corrected by adding more capabilities of the form **x*x***.

Other capabilities include **is**, an initialization string for the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to properly clear and then set the tabs on the terminal, if the terminal has settable tabs. If both are given, **is** will be printed before **if**. This is useful where **if** is */usr/lib/tabset/std* but **is** clears the tabs first.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **tc** can be given with the name of the similar terminal. This capability must be *last* and the combined length of the two entries must not exceed 1024. Since *termlib* routines search the entry from left to right, and since the tc capability is replaced by the corresponding entry, the capabilities given at the left override the ones in the similar terminal. A capability can be cancelled with **xx@** where xx is the capability. For example, the entry

```
hn|2621nl:ks@:ke@:tc=2621:
```

defines a 2621nl that does not have the **ks** or **ke** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

AUTHOR

Termcap is based on software developed by The University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

FILES

/etc/termcap	file containing terminal descriptions

SEE ALSO

ex(1), vi(1).

CAVEATS AND BUGS

Note *termcap* will be replaced by *terminfo* in the next release. Transition tools will be provided. *Ex* allows only 256 characters for string capabilities, and the routines in *termcap*(3) do not check for overflow of this buffer. The total length of a single entry (excluding only escaped newlines) may not exceed 1024.

The **ma**, **vs**, and **ve** entries are specific to the *vi* program.

Not all programs support all entries. There are entries that are not supported by any program.

NAME

/etc/utab – table of name neighbour UNIX United systems known to the Newcastle Connection at this system.

DESCRIPTION

/etc/utab contains one entry for each name neighbour of the system on which it is stored. Each entry consists of a 16-bit identifier (which must be in the range [0-255] for Release 1.0), a 16-bit length field whose value is the length of the following string plus one for the null byte, and a string which specifies the pathname of the name neighbour relative to the root is this system. The string is stored including the terminating null byte.

The "identifier" will be passed to your network interface routine "_netitoa()" when required to convert it to a physical address for your network. The inverse operation is performed by "_netatoi()", which returns an identifier given a physical address.

This file is maintained by the programs "mksys(8N)" and "rmsys(8N)", which can be used to inspect, add, modify, or delete an entry.

The file is used by the Newcastle Connection during "exec" processing to translate physical addresses (the 16-bit identifiers) into system names.

SEE ALSO

unite(8N), *mksys(8N)*, *rmsys(8N)*, "The Newcastle Connection – Release 1.0: Network Interface Installation Guide"

FILES

/etc/utab

NAME

utmp, wtmp – utmp and wtmp entry formats

SYNOPSIS

```
#include <sys/types.h>
#include <utmp.h>
```

DESCRIPTION

These files, which hold user and accounting information for such commands as *who*(1), *write*(1), and *login*(1), have the following structure as defined by <utmp.h>:

```
/*      <sys/types.h> must be included.                              */

#define UTMP_FILE       "/etc/utmp"
#define WTMP_FILE       "/etc/wtmp"
#define ut_name ut_user

struct utmp
  {
        char ut_user[8] ;              /* User login name */
        char ut_id[4] ;                /* /etc/lines id(usually line #) */
        char ut_line[12] ;             /* device name (console, lnxx) */
        short ut_pid ;                 /* process id */
        short ut_type ;                /* type of entry */
        struct exit_status
          {
            short e_termination ;      /* Process termination status */
            short e_exit ;             /* Process exit status */
          }
        ut_exit ;                      /* The exit status of a process
                                        * marked as DEAD_PROCESS.
                                        */

        time_t ut_time ;               /* time entry was made */
  } ;

/*      Definitions for ut_type                                      */

#define EMPTY           0
#define RUN_LVL         1
#define BOOT_TIME       2
#define OLD_TIME        3
#define NEW_TIME        4
#define INIT_PROCESS    5        /* Process spawned by "init" */
#define LOGIN_PROCESS   6        /* A "getty" process waiting for login */
#define USER_PROCESS    7        /* A user process */
#define DEAD_PROCESS    8
#define ACCOUNTING      9

#define UTMAXTYPE       ACCOUNTING      /* Largest legal value of ut_type */

/*      Special strings or formats used in the "ut_line" field when    */
/*      accounting for something other than a process.                 */
/*      No string for the ut_line field can be more than 11 chars +    */
/*      a NULL in length.                                              */

#define RUNLVL_MSG      "run-level %c"
#define BOOT_MSG        "system boot"
#define OTIME_MSG       "old time"
#define NTIME_MSG       "new time"
```

FILES

/usr/include/utmp.h
/etc/utmp
/etc/wtmp

SEE ALSO

login(1), who(1), write(1), getut(3C).

NAME

vfont – font formats for the Benson-Varian or Versatec

SYNOPSIS

/usr/lib/vfont/*

DESCRIPTION

The fonts for the printer/plotters have the following format. Each file contains a header, an array of 256 character description structures, and then the bit maps for the characters themselves. The header has the following format:

```
struct header {
        short   magic;
        unsigned short  size;
        short   maxx;
        short   maxy;
        short   xtnd;
} header;
```

The *magic* number is 0436 (octal). The *maxx*, *maxy*, and *xtnd* fields are not used at the current time. *Maxx* and *maxy* are intended to be the maximum horizontal and vertical size of any glyph in the font, in raster lines. The *size* is the size of the bit maps for the characters in bytes. Before the maps for the characters is an array of 256 structures for each of the possible characters in the font. Each element of the array has the form:

```
struct dispatch {
        unsigned short  addr;
        short   nbytes;
        char    up;
        char    down;
        char    left;
        char    right;
        short   width;
};
```

The *nbytes* field is nonzero for characters which actually exist. For such characters, the *addr* field is an offset into the rest of the file where the data for that character begins. There are *up+down* rows of data for each character, each of which has *left+right* bits, rounded up to a number of bytes. The *width* field is not used by vcat, although it is to make width tables for *troff*. It represents the logical width of the glyph, in raster lines, and shows where the base point of the next glyph would be.

FILES

/usr/lib/vfont/*

SEE ALSO

troff(1), pti(1), vfontinfo(1)

NAME

intro – introduction to games

DESCRIPTION

This section describes the recreational and educational programs found in the directory **/usr/games**.

NAME

arithmetic – provide drill in number facts

SYNOPSIS

/usr/games/arithmetic [**+–x/**] [range]

DESCRIPTION

Arithmetic types out simple arithmetic problems, and waits for an answer to be typed in. If the answer is correct, it types back "Right!", and a new problem. If the answer is wrong, it replies "What?", and waits for another answer. Every twenty problems, it publishes statistics on correctness and the time required to answer.

To quit the program, type an interrupt (delete).

The first optional argument determines the kind of problem to be generated; **+–x/** respectively cause addition, subtraction, multiplication, and division problems to be generated. One or more characters can be given; if more than one is given, the different types of problems will be mixed in random order; default is **+–**

Range is a decimal number; all addends, subtrahends, differences, multiplicands, divisors, and quotients will be less than or equal to the value of *range*. Default *range* is 10.

At the start, all numbers less than or equal to *range* are equally likely to appear. If the respondent makes a mistake, the numbers in the problem which was missed become more likely to reappear.

As a matter of educational philosophy, the program will not give correct answers, since the learner should, in principle, be able to calculate them. Thus the program is intended to provide drill for someone just past the first learning stage, not to teach number facts *de novo*. For almost all users, the relevant statistic should be time per problem, not percent correct.

NAME
back – the game of backgammon

SYNOPSIS
/usr/games/back

DESCRIPTION
Back is a program which provides a partner for the game of backgammon. It is designed to play at three different levels of skill, one of which you must select. In addition to selecting the opponent's level, you may also indicate that you would like to roll your own dice during your turns (for the superstitious players). You will also be given the opportunity to move first. The practice of each player rolling one die for the first move is not incorporated.

The points are numbered 1–24, with 1 being white's extreme inner table, 24 being brown's inner table, 0 being the bar for removed white pieces and 25 the bar for brown. For details on how moves are expressed, type **y** when *back* asks "Instructions?" at the beginning of the game. When *back* first asks "Move?", type **?** to see a list of move options other than entering your numerical move.

When the game is finished, *back* will ask you if you want the log. If you respond with **y**, *back* will attempt to append to or create a file **back.log** in the current directory.

FILES

/usr/games/lib/backrules	rules file
/tmp/b*	log temp file
back.log	log file

BUGS
The only level really worth playing is "expert", and it only plays the forward game.
Back will complain loudly if you attempt to make too *many* moves in a turn, but will become very silent if you make too *few*.
Doubling is not implemented.

NAME

backgammon – the game

SYNOPSIS

/usr/games/backgammon

DESCRIPTION

This program does what you expect. It will ask whether you need instructions.

NAME

banner – make long posters

SYNOPSIS

/usr/bin/banner

DESCRIPTION

Banner reads the standard input and prints it sideways in huge built-up letters on the standard output.

NAME

bcd, ppt — convert to antique media

SYNOPSIS

/usr/games/bcd text

/usr/games/ppt

DESCRIPTION

Bcd converts the literal *text* into a form familiar to old-timers.

Ppt converts the standard input into yet another form.

SEE ALSO

dd(1)

NAME

bj – the game of black jack

SYNOPSIS

/usr/games/bj

DESCRIPTION

Bj is a serious attempt at simulating the dealer in the game of black jack (or twenty-one) as might be found in Reno. The following rules apply:

The bet is $2 every hand.

> A player 'natural' (black jack) pays $3. A dealer natural loses $2. Both dealer and player naturals is a 'push' (no money exchange).
>
> If the dealer has an ace up, the player is allowed to make an 'insurance' bet against the chance of a dealer natural. If this bet is not taken, play resumes as normal. If the bet is taken, it is a side bet where the player wins $2 if the dealer has a natural and loses $1 if the dealer does not.
>
> If the player is dealt two cards of the same value, he is allowed to 'double'. He is allowed to play two hands, each with one of these cards. (The bet is doubled also; $2 on each hand.)
>
> If a dealt hand has a total of ten or eleven, the player may 'double down'. He may double the bet ($2 to $4) and receive exactly one more card on that hand.
>
> Under normal play, the player may 'hit' (draw a card) as long as his total is not over twenty-one. If the player 'busts' (goes over twenty-one), the dealer wins the bet.
>
> When the player 'stands' (decides not to hit), the dealer hits until he attains a total of seventeen or more. If the dealer busts, the player wins the bet.
>
> If both player and dealer stand, the one with the largest total wins. A tie is a push.

The machine deals and keeps score. The following questions will be asked at appropriate times. Each question is answered by **y** followed by a new line for 'yes', or just new line for 'no'.

?	(means, 'do you want a hit?')
Insurance?	
Double down?	

Every time the deck is shuffled, the dealer so states and the 'action' (total bet) and 'standing' (total won or lost) is printed. To exit, hit the interrupt key (CTRL-C) and the action and standing will be printed.

NAME

craps – the game of craps

SYNOPSIS

/usr/games/craps

DESCRIPTION

Craps is a form of the game of craps that is played in Las Vegas. The program simulates the *roller*, while the user (the *player*) places bets. The player may choose, at any time, to bet with the roller or with the *House*. A bet of a negative amount is taken as a bet with the House, any other bet is a bet with the roller.

The player starts off with a "bankroll" of $2,000.

The program prompts with:

```
bet?
```

The bet can be all or part of the player's bankroll. Any bet over the total bankroll is rejected and the program prompts with "bet?" until a proper bet is made.

Once the bet is accepted, the roller throws the dice. The following rules apply (the player wins or loses depending on whether the bet is placed with the roller or with the House; the odds are even). The *first* roll is the roll immediately following a bet.

1. On the first roll:

7 or 11	wins for the roller;
2, 3, or 12	wins for the House;
any other number	is the *point*, roll again (Rule 2 applies).

2. On subsequent rolls:

point	roller wins;
7	House wins;
any other number	roll again.

If a player loses the entire bankroll, the House will offer to lend the player an additional $2,000. The program will prompt:

```
marker?
```

A "yes" (or "y") consummates the loan. Any other reply terminates the game.

If a player owes the House money, the House reminds the player, before a bet is placed, how many markers are outstanding.

If, at any time, the bankroll of a player who has outstanding markers exceeds $2,000, the House asks:

```
Repay marker?
```

A reply of "yes" (or "y") indicates the player's willingness to repay the loan. If only 1 marker is outstanding, it is immediately repaid. However, if more than 1 marker are outstanding, the House asks:

```
        How many?
```

markers the player would like to repay. If an invalid number is entered (or just a carriage return), an appropriate message is printed and the program will prompt with "How many?" until a valid number is entered.

If a player accumulates 10 markers (a total of $20,000 borrowed from the House), the program informs the player of the situation and exits.

Should the bankroll of a player who has outstanding markers exceed $50,000, the *total* amount of money borrowed will be *automatically* repaid to the House.

Any player who accumulates $100,000 or more breaks the bank. The program then prompts:

```
        New game?
```

to give the House a chance to win back its money.

Any reply other than "yes" is considered "no" (except in the case of "bet?" or "How many?"). To exit, send an interrupt (break), DEL, or control-D. The program will indicate whether the player won, lost, or broke even.

MISCELLANEOUS

The random number generator for the die numbers uses the seconds from the time of day. Depending on system usage, these numbers, at times, may seem strange but occurrences of this type in a real dice situation are not uncommon.

NAME

hangman – guess the word

SYNOPSIS

/usr/games/hangman [arg]

DESCRIPTION

Hangman chooses a word at least seven letters long from a dictionary. The user is to guess letters one at a time.

The optional argument *arg* names an alternate dictionary.

FILES

/usr/dict/words

BUGS

Hyphenated compounds are run together.

NAME
    moo – guessing game

SYNOPSIS
    /usr/games/moo

DESCRIPTION
    Moo is a guessing game imported from England. The computer picks a number consisting of four distinct decimal digits. The player guesses four distinct digits being scored on each guess. A 'cow' is a correct digit in an incorrect position. A 'bull' is a correct digit in a correct position. The game continues until the player guesses the number (a score of four bulls).

NAME

quiz – test your knowledge

SYNOPSIS

/usr/games/quiz [**–i** file] [**–t**] [category1 category2]

DESCRIPTION

Quiz gives associative knowledge tests on various subjects. It asks items chosen from *category1* and expects answers from *category2*. If no categories are specified, *quiz* gives instructions and lists the available categories.

Quiz tells a correct answer whenever you type a bare newline. At the end of input, upon interrupt, or when questions run out, *quiz* reports a score and terminates.

The **–t** flag specifies 'tutorial' mode, where missed questions are repeated later, and material is gradually introduced as you learn.

The **–i** flag causes the named file to be substituted for the default index file. The lines of these files have the syntax:

```
line      = category newline | category ':' line
category  = alternate | category '|' alternate
alternate = empty | alternate primary
primary   = character | '[' category ']' | option
option    = '{' category '}'
```

The first category on each line of an index file names an information file. The remaining categories specify the order and contents of the data in each line of the information file. Information files have the same syntax. Backslash '\' is used as with *sh*(1) to quote syntactically significant characters or to insert transparent newlines into a line. When either a question or its answer is empty, *quiz* will refrain from asking it.

FILES

/usr/games/quiz.k/*

BUGS

The construct 'a|ab' doesn't work in an information file. Use 'a{b}'.

NAME

reversi – reversi, a game of dramatic reversals

SYNOPSIS

/usr/games/reversi [B] [**b#**] [**d#**] [IBmv] [IWmv] [ifile] [**l#**] [ofile] [q] [r[#]] [**s#**] [T] *file*]

DESCRIPTION

Reversi, (a.k.a *othello*, a.k.a *oxo*), is played on an square board, (usually 8 x 8), using tokens which are "white", (O), on one side, and "black", (X), on the other. Each player takes his turn by placing a token with his color up in an empty square. The board initially contains two "O" and two "X" tokens. With each turn, a player must flip over one or more tokens displaying his opponent's color. He does this by placing one of his tokens such that he outflanks one or more of his opponent's, horizontally, vertically, or diagonally. The outflanked tokens are flipped over and thus can be re-flipped. If a player cannot outflank his opponent, he must pass thereby forfeiting his turn. The play continues until both players must pass.

In this game you move by typing in the column letter and row number at which you want to place your token. You can also type in:

?	to re-draw the board,
~n	to retract your last move, (handy for cheating),
pass	to acknowledge that you have no legal move,
resign	to give up, and
!	to escape to the Shell.

Reversi has several flag arguments. Their meanings are:

B	The computer plays "black" and goes first.
b#	The board size is set to **#**x**#**, (max is 10x10). The default is 8x8.
d#	The debug flag is turned on; **#** indicates how much meaningless trace information you'd like to be buried under. This flag also forces the T flag.
IBmv	Initialize the square at "mv" to hold a black token, where "mv" is the letter-number of a square on the board. This is useful for starting a game with an arbitrary board configuration.
IWmv	Initialize the square at "mv" to hold a white token, as above.
ifoo	Take move input from the file named "foo". Useful for having the program play against other programs, (this sentence no verb).
l#	Set look-ahead level to **#** initially; look-ahead level is modified dynamically to try for a given compute time per move, (see the **s** flag, below).
ofubar	Send computer moves to the file named "fubar". The format is the same as expected for input, (see i flag, above).

q	Quiet mode. Suppress gratuitous display of the board.
r#	Report on look-ahead results down to level #. This option is similar, but not identical, to **d**. This flag also forces the **T** flag.
s#	Attempt to use # seconds of combined user and system time for each computer move. If unspecified the default is 5 seconds.
T	The terminal in use either has no cursor addressing or has different cursor addressing from the standard. Normally, the playing board is displayed and modified on the screen via cursor motion commands and the list of moves is scrolled at the bottom of the screen. This flag indicates that your terminal is functionally a teletype and should be treated as such.

DIAGNOSTICS

Fairly reasonable explanations of illegal moves, etc.

NAME

rogue – Exploring The Dungeons of Doom

SYNOPSIS

/usr/games/rogue [**−r**] [*save_file*] [**−s**] [**−d**]

DESCRIPTION

Rogue is a computer fantasy game with a new twist. It is crt oriented and the object of the game is to survive the attacks of various monsters and get a lot of gold, rather than the puzzle solving orientation of most computer fantasy games.

To get started you really only need to know two commands. The command ? will give you a list of the available commands and the command / will identify the things you see on the screen.

To win the game (as opposed to merely playing to beat other people high scores) you must locate the Amulet of Yendor which is somewhere below the 20th level of the dungeon and get it out. Nobody has achieved this yet and if somebody does, they will probably go down in history as a hero among heros.

When the game ends, either by your death, when you quit, or if you (by some miracle) manage to win, *rogue* will give you a list of the top-ten scorers. The scoring is based entirely upon how much gold you get. There is a 10% penalty for getting yourself killed.

If *save_file* is specified, rogue will be restored from the specified saved game file. If the **−r** option is used, the save game file is presumed to be the default.

The **−s** option will print out the list of scores.

The **−d** option will kill you and try to add you to the score file.

For more detailed directions, read the document *A Guide to the Dungeons of Doom.*

AUTHORS

Michael C. Toy, Kenneth C. R. C. Arnold, Glenn Wichman

FILES

/usr/games/lib/.rogue_roll	Score file
~/.rogue.save	Default save file

SEE ALSO

Michael C. Toy and Kenneth C. R. C. Arnold, *A guide to the Dungeons of Doom*

BUGS

Probably infinite. However, that Ice Monsters sometimes transfix you permanently is *not* a bug. It's a feature.

NAME

startrek – THE game based on the t.v. series.

SYNOPSIS

/usr/games/startrek

DESCRIPTION

You are the captain of the starship Enterprise and you have to destroy a random number of klingons (typically 15-25) in 30 stardates. (A measure of time in space, think of it as a day.) Full instructions are given if you reply 'y' to DO YOU WANT INSTRUCTIONS? A brief list of instructions is given if you ever type in an illegal command.

If you reply 'p' to PILOT TRAINING OR REAL MISSION? the computer asks you for a task number. This is used to start the random number generator so you can play in the same galaxy again if you want to.

Docking at a starbase refuels and rearms the Enterprise. If you stop for repairs you are delayed one stardate. Waiting for repairs in space might also cost you time.

BUGS

The calculator returns distances slightly too large for inter-quadrant travel.

FILES

```
/usr/games/startrek  object code
/usr/lib/startrek    instructions
```

AUTHOR

Originally written in Basic by Mike Mayfield, Centreline Engineering and extended by David Ahl of Creative Computing.
Translated into C and extended by M.J.Bayliss UKC April-October 1977.

NAME

ttt − tic-tac-toe

SYNOPSIS

/usr/games/ttt

DESCRIPTION

Ttt is the X and O game popular in the first grade. This is a learning program that never makes the same mistake twice.

Although it learns, it learns slowly. It must lose nearly 80 games to completely know the game.

FILES

ttt.a	learning file

NAME

wump – the game of hunt-the-wumpus

SYNOPSIS

/usr/games/wump

DESCRIPTION

Wump plays the game of 'Hunt the Wumpus.' A Wumpus is a creature that lives in a cave with several rooms connected by tunnels. You wander among the rooms, trying to shoot the Wumpus with an arrow, meanwhile avoiding being eaten by the Wumpus and falling into Bottomless Pits. There are also Super Bats which are likely to pick you up and drop you in some random room.

The program asks various questions which you answer one per line; it will give a more detailed description if you want.

This program is based on one described in *People's Computer Company*, *2*, 2 (November 1973).

BUGS

It will never replace Space War.

1. Introduction

You have just finished your years as a student at the local fighter's guild. After much practice and sweat you have finally completed your training and are ready to embark upon a perilous adventure. As a test of your skills, the local guildmasters have sent you into the Dungeons of Doom. Your task is to return with the Amulet of Yendor. Your reward for the completion of this task will be a full membership in the local guild. In addition, you are allowed to keep all the loot you bring back from the dungeons.

In preparation for your journey, you are given an enchanted mace, a bow, and a quiver of arrows taken from a dragon's hoard in the far off Dark Mountains. You are also outfitted with elf-crafted armor and given enough food to reach the dungeons. You say goodbye to family and friends for what may be the last time and head up the road.

You set out on your way to the dungeons and after several days of uneventful travel, you see the ancient ruins that mark the entrance to the Dungeons of Doom. It is late at night, so you make camp at the entrance and spend the night sleeping under the open skies. In the morning you gather your weapons, put on your armor, eat what is almost your last food, and enter the dungeons.

2. What is going on here?

You have just begun a game of rogue. Your goal is to grab as much treasure as you can, find the Amulet of Yendor, and get out of the Dungeons of Doom alive. On the screen, a map of where you have been and what you have seen on the current dungeon level is kept. As you explore more of the level, it appears on the screen in front of you.

Rogue differs from most computer fantasy games in that it is screen oriented. Commands are all one or two keystrokes[1] and the results of your commands are displayed graphically on the screen rather than being explained in words.[2]

Another major difference between rogue and other computer fantasy games is that once you have solved all the puzzles in a standard fantasy game, it has lost most of its excitement and it ceases to be fun. Rogue, on the other

[1] As opposed to pseudo English sentences.

[2] A minimum screen size of 24 lines by 80 columns is required. If the screen is larger, only the 24x80 section will be used for the map.

hand, generates a new dungeon every time you play it and even the author finds it an entertaining and exciting game.

3. What do all those things on the screen mean?

In order to understand what is going on in rogue you have to first get some grasp of what rogue is doing with the screen. The rogue screen is intended to replace the "You can see ..." descriptions of standard fantasy games. Figure 1 is a sample of what a rogue screen might look like.

3.1. The bottom line

At the bottom line of the screen are a few pieces of cryptic information describing your current status. Here is an explanation of what these things mean:

Level	This number indicates how deep you have gone in the dungeon. It starts at one and goes up as you go deeper into the dungeon.
Gold	The number of gold pieces you have managed to find and keep with you so far.
Hp	Your current and maximum health points. Health points indicate how much damage you can take before you die. The more you get hit in a fight, the lower they get. You can regain health points by resting. The number in parentheses is the maximum number your health points can reach.
Str	Your current strength and maximum ever strength. This can be any integer less than or equal to 31, or

```
                    ------------
                    |..........+
                    |..@....]..|
                    |....B.....|
                    |..........|
                    -----+------



Level: 1  Gold: 0      Hp: 12(12)  Str: 16(16)  Arm: 4  Exp: 1/0
```

Figure 1

greater than or equal to three. The higher the number, the stronger you are. The number in the parentheses is the maximum strength you have attained so far this game.

Arm Your current armor protection. This number indicates how effective your armor is in stopping blows from unfriendly creatures. The higher this number is, the more effective the armor.

Exp These two numbers give your current experience level and experience points. As you do things, you gain experience points. At certain experience point totals, you gain an experience level. The more experienced you are, the better you are able to fight and to withstand magical attacks.

3.2. The top line

The top line of the screen is reserved for printing messages that describe things that are impossible to represent visually. If you see a "--More--" on the top line, this means that rogue wants to print another message on the screen, but it wants to make certain that you have read the one that is there first. To read the next message, just type a space.

3.3. The rest of the screen

The rest of the screen is the map of the level as you have explored it so far. Each symbol on the screen represents something. Here is a list of what the various symbols mean:

	This symbol represents you, the adventurer.
-|	These symbols represent the walls of rooms.
+	A door to/from a room.
.	The floor of a room.
#	The floor of a passage between rooms.
*	A pile or pot of gold.
)	A weapon of some sort.
]	A piece of armor.
!	A flask containing a magic potion.

?	A piece of paper, usually a magic scroll.
=	A ring with magic properties
/	A magical staff or wand
^	A trap, watch out for these.
%	A staircase to other levels
:	A piece of food.
A-Z	The uppercase letters represent the various inhabitants of the Dungeons of Doom. Watch out, they can be nasty and vicious.

4. Commands

Commands are given to rogue by typing one or two characters. Most commands can be preceded by a count to repeat them (e.g. typing "10s" will do ten searches). Commands for which counts make no sense have the count ignored. To cancel a count or a prefix, type <ESCAPE>. The list of commands is rather long, but it can be read at any time during the game with the "?" command. Here it is for reference, with a short explanation of each command.

?   The help command. Asks for a character to give help on. If you type a "*", it will list all the commands, otherwise it will explain what the character you typed does.

/   This is the "What is that on the screen?" command. A "/" followed by any character that you see on the level, will tell you what that character is. For instance, typing "/@" will tell you that the "@" symbol represents you, the player.

h, H, ^H
Move left. You move one space to the left. If you use upper case "h", you will continue to move left until you run into something. This works for all movement commands (e.g. "L" means run in direction "l") If you use the "control" "h", you will continue moving in the specified direction until you pass something interesting or run into a wall. You should experiment with this, since it is a very useful command, but very difficult to describe. This also works for all movement commands.

j   Move down.

k	Move up.
l	Move right.
y	Move diagonally up and left.
u	Move diagonally up and right.
b	Move diagonally down and left.
n	Move diagonally down and right.
t	Throw an object. This is a prefix command. When followed with a direction it throws an object in the specified direction. (e.g. type "th" to throw something to the left.)
f	Fight until someone dies. When followed with a direction this will force you to fight the creature in that direction until either you or it bites the big one.
m	Move onto something without picking it up. This will move you one space in the direction you specify and, if there is an object there you can pick up, it won't do it.
z	Zap prefix. Point a staff or wand in a given direction and fire it. Even non-directional staves must be pointed in some direction to be used.
^	Identify trap command. If a trap is on your map and you can't remember what type it is, you can get rogue to remind you by getting next to it and typing "^" followed by the direction that would move you on top of it.
s	Search for traps and secret doors. Examine each space immediately adjacent to you for the existence of a trap or secret door. There is a large chance that even if there is something there, you won't find it, so you might have to search a while before you find something.
>	Climb down a staircase to the next level. Not surprisingly, this can only be done if you are standing on staircase.
<	Climb up a staircase to the level above. This can't be done without the Amulet of Yendor in your possession.
.	Rest. This is the "do nothing" command. This is good for waiting and healing.

*	Inventory. List what you are carrying in your pack.
I	Selective inventory. Tells you what a single item in your pack is.
q	Quaff one of the potions you are carrying.
r	Read one of the scrolls in your pack.
e	Eat food from your pack.
w	Wield a weapon. Take a weapon out of your pack and carry it for use in combat, replacing the one you are currently using (if any).
W	Wear armor. You can only wear one suit of armor at a time. This takes extra time.
T	Take armor off. You can't remove armor that is cursed. This takes extra time.
P	Put on a ring. You can wear only two rings at a time (one on each hand). If you aren't wearing any rings, this command will ask you which hand you want to wear it on, otherwise, it will place it on the unused hand. The program assumes that you wield your sword in your right hand.
R	Remove a ring. If you are only wearing one ring, this command takes it off. If you are wearing two, it will ask you which one you wish to remove,
d	Drop an object. Take something out of your pack and leave it lying on the floor. Only one object can occupy each space. You cannot drop a cursed object at all if you are wielding or wearing it.
c	Call an object something. If you have a type of object in your pack which you wish to remember something about, you can use the call command to give a name to that type of object. This is usually used when you figure out what a potion, scroll, ring, or staff is after you pick it up, or when you want to remember which of those swords in your pack you were wielding.
D	Print out which things you've discovered something about. This command will ask you what type of thing you are interested in. If you type the character for a given type of object (e.g. "!" for potion) it will tell you which kinds of that type of object you've discovered (i.e., figured out what they are). This command works for potions, scrolls, rings, and staves and wands.

o Examine and set options. This command is further explained in the section on options.

^R Redraws the screen. Useful if spurious messages or transmission errors have messed up the display.

^P Print last message. Useful when a message disappears before you can read it. This only repeats the last message that was not a mistyped command so that you don't loose anything by accidentally typing the wrong character instead of ^P.

<ESCAPE>
Cancel a command, prefix, or count.

! Escape to a shell for some commands.

Q Quit. Leave the game.

S Save the current game in a file. It will ask you whether you wish to use the default save file. Caveat: Rogue won't let you start up a copy of a saved game, and it removes the save file as soon as you start up a restored game. This is to prevent people from saving a game just before a dangerous position and then restarting it if they die. To restore a saved game, give the file name as an argument to rogue. As in

```
        % rogue save_file
```

To restart from the default save file (see below), run

```
        % rogue -r
```

v Prints the program version number.

) Print the weapon you are currently wielding

] Print the armor you are currently wearing

= Print the rings you are currently wearing

Reprint the status line on the message line

5. Rooms

Rooms in the dungeons are either lit or dark. If you walk into a lit room, the entire room will be drawn on the screen as soon as you enter. If you walk into a dark room, it will only be displayed as you explore it. Upon leaving a room, all monsters inside the room are erased from the screen. In the darkness you can only see one space in all directions around you. A corridor is always dark.

6. Fighting

If you see a monster and you wish to fight it, just attempt to run into it. Many times a monster you find will mind its own business unless you attack it. It is often the case that discretion is the better part of valor.

7. Objects you can find

When you find something in the dungeon, it is common to want to pick the object up. This is accomplished in rogue by walking over the object (unless you use the "m" prefix, see above). If you are carrying too many things, the program will tell you and it won't pick up the object, otherwise it will add it to your pack and tell you what you just picked up.

Many of the commands that operate on objects must prompt you to find out which object you want to use. If you change your mind and don't want to do that command after all, just type an <ESCAPE> and the command will be aborted.

Some objects, like armor and weapons, are easily differentiated. Others, like scrolls and potions, are given labels which vary according to type. During a game, any two of the same kind of object with the same label are the same type. However, the labels will vary from game to game.

When you use one of these labeled objects, if its effect is obvious, rogue will remember what it is for you. If it's effect isn't extremely obvious you will be asked what you want to scribble on it so you will recognize it later, or you can use the "call" command (see above).

7.1. Weapons

Some weapons, like arrows, come in bunches, but most come one at a time. In order to use a weapon, you must wield it. To fire an arrow out of a bow, you must first wield the bow, then throw the arrow. You can only wield one weapon at a time, but you can't change weapons if the one you are currently wielding is cursed. The commands to use weapons are "w" (wield) and "t" (throw).

7.2. Armor

There are various sorts of armor lying around in the dungeon. Some of it is enchanted, some is cursed, and some is just normal. Different armor types have different armor protection. The higher the armor protection, the more protection the armor affords against the blows of monsters. Here is a list of the various armor types and their normal armor protection:

Type	Protection
None	0
Leather armor	2
Studded leather / Ring mail	3
Scale mail	4
Chain mail	5
Banded mail / Splint mail	6

If a piece of armor is enchanted, its armor protection will be higher than normal. If a suit of armor is cursed, its armor protection will be lower, and you will not be able to remove it. However, not all armor with a protection that is lower than normal is cursed.

The commands to use weapons are "W" (wear) and "T" (take off).

7.3. Scrolls

Scrolls come with titles in an unknown tongue[3]. After you read a scroll, it disappears from your pack. The command to use a scroll is "r" (read).

7.4. Potions

Potions are labeled by the color of the liquid inside the flask. They disappear after being quaffed. The command to use a scroll is "q" (quaff).

7.5. Staves and Wands

Staves and wands do the same kinds of things. Staves are identified by a type of wood; wands by a type of metal or bone. They are generally things you want to do to something over a long distance, so you must point them at what you wish to affect to use them. Some staves are not affected by the direction they are pointed, though. Staves come with multiple magic charges, the number being random, and when they are used up, the staff is just a piece of wood or metal.

---

[3] Actually, it's a dialect spoken only by the twenty-seven members of a tribe in Outer Mongolia, but you're not supposed to know that.

The command to use a wand or staff is "z" (zap)

7.6. Rings

Rings are very useful items, since they are relatively permanent magic, unlike the usually fleeting effects of potions, scrolls, and staves. Of course, the bad rings are also more powerful. Most rings also cause you to use up food more rapidly, the rate varying with the type of ring. Rings are differentiated by their stone settings. The commands to use rings are "P" (put on) and "R" (remove).

7.7. Food

Food is necessary to keep you going. If you go too long without eating you will faint, and eventually die of starvation. The command to use food is "e" (eat).

8. Options

Due to variations in personal tastes and conceptions of the way rogue should do things, there are a set of options you can set that cause rogue to behave in various different ways.

8.1. Setting the options

There are two ways to set the options. The first is with the "o" command of rogue; the second is with the "ROGUEOPTS" environment variable[4].

8.1.1. Using the 'o' command

When you type "o" in rogue, it clears the screen and displays the current settings for all the options. It then places the cursor by the value of the first option and waits for you to type. You can type a <RETURN> which means to go to the next option, a "-" which means to go to the previous option, an <ESCAPE> which means to return to the game, or you can give the option a value. For boolean options this merely involves typing "t" for true or "f" for false. For string options, type the new value followed by a <RETURN>.

8.1.2. Using the ROGUEOPTS variable

The ROGUEOPTS variable is a string containing a comma separated list of initial values for the various options. Boolean variables can be turned on by listing their name or

[4] On Version 6 systems, there is no equivalent of the ROGUEOPTS feature.

turned off by putting a "no" in front of the name. Thus to set up an environment variable so that jump is on, terse is off, and the name is set to "Blue Meanie", use the command

```
   % setenv ROGUEOPTS "jump,noterse,name=Blue Meanie"[5]
```

8.2. Option list

Here is a list of the options and an explanation of what each one is for. The default value for each is enclosed in square brackets. For character string options, input over fifty characters will be ignored.

terse [noterse]
: Useful for those who are tired of the sometimes lengthy messages of rogue. This is a useful option for playing on slow terminals, so this option defaults to terse if you are on a slow (1200 baud or under) terminal.

jump [nojump]
: If this option is set, running moves will not be displayed until you reach the end of the move. This saves considerable cpu and display time. This option defaults to jump if you are using a slow terminal.

flush [noflush]
: All typeahead is thrown away after each round of battle. This is useful for those who type far ahead and then watch in dismay as a Bat kills them.

seefloor [seefloor]
: Display the floor around you on the screen as you move through dark rooms. Due to the amount of characters generated, this option defaults to noseefloor if you are using a slow terminal.

passgo [nopassgo]
: Follow turnings in passageways. If you run in a passage and you run into stone or a wall, rogue will see if it can turn to the right or left. If it can only turn one way, it will turn that way. If it can turn either or neither, it will stop. This is followed strictly, which can sometimes lead to slightly confusing occurrences (which is why it defaults to nopassgo).

tombstone [tombstone]
: Print out the tombstone at the end if you get killed.

---

[5] For those of you who use the bourne shell, the commands would be

```
   $ ROGUEOPTS="jump,noterse,name=Blue Meanie"
   $ export ROGUEOPTS
```

This is nice but slow, so you can turn it off if you like.

inven [overwrite]
Inventory type. This can have one of three values: overwrite, slow, or clear. With overwrite the top lines of the map are overwritten with the list when inventory is requested or when "Which item do you wish to . . .? " questions are answered with a "*". However, if the list is longer than a screenful, the screen is cleared. With slow, lists are displayed one item at a time on the top of the screen, and with clear, the screen is cleared, the list is displayed, and then the dungeon level is re-displayed. Due to speed considerations, clear is the default for terminals without clear-to-end-of-line capabilities.

name [account name]
This is the name of your character. It is used if you get on the top ten scorer's list.

fruit [slime-mold]
This should hold the name of a fruit that you enjoy eating. It is basically a whimsey that rogue uses in a couple of places.

file [~/rogue.save]
The default file name for saving the game. If your phone is hung up by accident, rogue will automatically save the game in this file. The file name may start with the special character "~" which expands to be your home directory.

9. Scoring

Rogue usually maintains a list of the top scoring people or scores on your machine. Depending on how it is set up, it can post either the top scores or the top players. In the latter case, each account on the machine can post only one non-winning score on this list. If you score higher than someone else on this list, or better your previous score on the list, you will be inserted in the proper place under your current name. How many scores are kept can also be set up by whoever installs it on your machine.

If you quit the game, you get out with all of your gold intact. If, however, you get killed in the Dungeons of Doom, your body is forwarded to your next-of-kin, along with 90% of your gold; ten percent of your gold is kept by the

Dungeons' wizard as a fee[6]. This should make you consider whether you want to take one last hit at that monster and possibly live, or quit and thus stop with whatever you have. If you quit, you do get all your gold, but if you swing and live, you might find more.

If you just want to see what the current top players/games list is, you can type

```
        % rogue -s
```

10. Acknowledgements

Rogue was originally conceived of by Glenn Wichman and Michael Toy. Ken Arnold and Michael Toy then smoothed out the user interface, and added jillions of new features. We would like to thank Bob Arnold, Michelle Busch, Andy Hatcher, Kipp Hickman, Mark Horton, Daniel Jensen, Bill Joy, Joe Kalash, Steve Maurer, Marty McNary, Jan Miller, and Scott Nelson for their ideas and assistance; and also the teeming multitudes who graciously ignored work, school, and social life to play rogue and send us bugs, complaints, suggestions, and just plain flames. And also Mom.

---

[6] The Dungeon's wizard is named Wally the Wonder Badger. Invocations should be accompanied by a sizable donative.

NAME

intro – introduction to miscellany

DESCRIPTION

This section describes miscellaneous facilities such as macro packages, character set tables, etc.

NAME

ascii – map of ASCII character set

SYNOPSIS

```
cat /usr/pub/ascii
```

DESCRIPTION

Ascii is a map of the ASCII character set, giving both octal and hexadecimal equivalents of each character, to be printed as needed. It contains:

```
                               ASCII hex
-------------------------------------------------------------------------
! 00 nul ! 01 soh ! 02 stx ! 03 etx ! 04 eot ! 05 enq ! 06 ack ! 07 bel !
! 08 bs  ! 09 ht  ! 0A nl  ! 0B vt  ! 0C np  ! 0D cr  ! 0E so  ! 0F si  !
! 10 dle ! 11 dc1 ! 12 dc2 ! 13 dc3 ! 14 dc4 ! 15 nak ! 16 syn ! 17 etb !
! 18 can ! 19 em  ! 1A sub ! 1B esc ! 1C fs  ! 1D gs  ! 1E rs  ! 1F us  !
! 20 sp  ! 21  !  ! 22  "  ! 23  #  ! 24  $  ! 25  %  ! 26  &  ! 27  '  !
! 28  (  ! 29  )  ! 2A  *  ! 2B  +  ! 2C  ,  ! 2D  -  ! 2E  .  ! 2F  /  !
! 30  0  ! 31  1  ! 32  2  ! 33  3  ! 34  4  ! 35  5  ! 36  6  ! 37  7  !
! 38  8  ! 39  9  ! 3A  :  ! 3B  ;  ! 3C  <  ! 3D  =  ! 3E  >  ! 3F  ?  !
! 40  @  ! 41  A  ! 42  B  ! 43  C  ! 44  D  ! 45  E  ! 46  F  ! 47  G  !
! 48  H  ! 49  I  ! 4A  J  ! 4B  K  ! 4C  L  ! 4D  M  ! 4E  N  ! 4F  O  !
! 50  P  ! 51  Q  ! 52  R  ! 53  S  ! 54  T  ! 55  U  ! 56  V  ! 57  W  !
! 58  X  ! 59  Y  ! 5A  Z  ! 5B  [  ! 5C  \  ! 5D  ]  ! 5E  ^  ! 5F  _  !
! 60  `  ! 61  a  ! 62  b  ! 63  c  ! 64  d  ! 65  e  ! 66  f  ! 67  g  !
! 68  h  ! 69  i  ! 6A  j  ! 6B  k  ! 6C  l  ! 6D  m  ! 6E  n  ! 6F  o  !
! 70  p  ! 71  q  ! 72  r  ! 73  s  ! 74  t  ! 75  u  ! 76  v  ! 77  w  !
! 78  x  ! 79  y  ! 7A  z  ! 7B  {  ! 7C  |  ! 7D  }  ! 7E  ~  ! 7F del !
-------------------------------------------------------------------------

                               ASCII octal
-------------------------------------------------------------------------
! 000 nul! 001 soh! 002 stx! 003 etx! 004 eot! 005 enq! 006 ack! 007 bel!
! 010 bs ! 011 ht ! 012 nl ! 013 vt ! 014 np ! 015 cr ! 016 so ! 017 si !
! 020 dle! 021 dc1! 022 dc2! 023 dc3! 024 dc4! 025 nak! 026 syn! 027 etb!
! 030 can! 031 em ! 032 sub! 033 esc! 034 fs ! 035 gs ! 036 rs ! 037 us !
! 040 sp ! 041 !  ! 042 "  ! 043 #  ! 044 $  ! 045 %  ! 046 &  ! 047 '  !
! 050 (  ! 051 )  ! 052 *  ! 053 +  ! 054 ,  ! 055 -  ! 056 .  ! 057 /  !
! 060 0  ! 061 1  ! 062 2  ! 063 3  ! 064 4  ! 065 5  ! 066 6  ! 067 7  !
! 070 8  ! 071 9  ! 072 :  ! 073 ;  ! 074 <  ! 075 =  ! 076 >  ! 077 ?  !
! 100 @  ! 101 A  ! 102 B  ! 103 C  ! 104 D  ! 105 E  ! 106 F  ! 107 G  !
! 110 H  ! 111 I  ! 112 J  ! 113 K  ! 114 L  ! 115 M  ! 116 N  ! 117 O  !
! 120 P  ! 121 Q  ! 122 R  ! 123 S  ! 124 T  ! 125 U  ! 126 V  ! 127 W  !
! 130 X  ! 131 Y  ! 132 Z  ! 133 [  ! 134 \  ! 135 ]  ! 136 ^  ! 137 _  !
! 140 `  ! 141 a  ! 142 b  ! 143 c  ! 144 d  ! 145 e  ! 146 f  ! 147 g  !
! 150 h  ! 151 i  ! 152 j  ! 153 k  ! 154 l  ! 155 m  ! 156 n  ! 157 o  !
! 160 p  ! 161 q  ! 162 r  ! 163 s  ! 164 t  ! 165 u  ! 166 v  ! 167 w  !
! 170 x  ! 171 y  ! 172 z  ! 173 {  ! 174 |  ! 175 }  ! 176 ~  ! 177 del!
-------------------------------------------------------------------------
```

FILES

/usr/pub/ascii

NAME

environ – user environment

DESCRIPTION

An array of strings called the "environment" is made available by *exec*(2) when a process begins. By convention, these strings have the form "name=value". The following names are used by various commands:

PATH The sequence of directory prefixes that *sh*(1), *time*(1), *nice*(1), *nohup*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by colons (:). *Login*(1) sets **PATH=:/usr/ucb:/bin:/usr/bin:/usr/local**

HOME Name of the user's login directory, set by *login*(1) from the password file *passwd*(5).

TERM The kind of terminal for which output is to be prepared. This information is used by commands, such as *mm*(1) or *tplot*(1G), which may exploit special capabilities of that terminal.

TZ Time zone information. The format is **xxx***n***zzz** where **xxx** is standard local time zone abbreviation, *n* is the difference in hours from GMT, and **zzz** is the abbreviation for the daylight-saving local time zone, if any; for example, **EST5EDT**.

Further names may be placed in the environment by the *export* command and "name=value" arguments in *sh*(1), or by *exec*(2). It is unwise to conflict with certain shell variables that are frequently exported by .profile files: **MAIL**, **PS1**, **PS2**, **IFS**.

SEE ALSO

env(1), login(1), sh(1), exec(2), getenv(3C), profile(5), term(7).

NAME

eqnchar – special character definitions for eqn and neqn

SYNOPSIS

eqn /usr/pub/eqnchar [files] **| troff** [options]

neqn /usr/pub/eqnchar [files] **| nroff** [options]

DESCRIPTION

Eqnchar contains *troff*(1) and *nroff*(1) character definitions for constructing characters that are not available on the Wang Laboratories, Inc. C/A/T phototypesetter. These definitions are primarily intended for use with *eqn*(1) and *neqn*(1); *eqnchar* contains definitions for the following characters:

ciplus	⊕	*||*	‖	*square*	▪
citimes	⊗	*langle*	〈	*circle*	○
wig	~	*rangle*	〉	*blot*	■
-wig	≃	*hbar*	ℏ	*bullet*	•
>wig	≳	*ppd*	⊥	*prop*	∝
<wig	≲	*<->*	↔	*empty*	∅
=wig	≅	*<=>*	⇔	*member*	∈
star	*	*|<*	≮	*nomem*	∉
bigstar	✱	*|>*	≯	*cup*	∪
=dot	≐	*ang*	∠	*cap*	∩
orsign	∨	*rang*	∟	*incl*	⊑
andsign	∧	*3dot*	⋮	*subset*	⊂
=del	≜	*thf*	∴	*supset*	⊃
oppA	∀	*quarter*	¼	*!subset*	⊆
oppE	∃	*3quarter*	¾	*!supset*	⊇
angstrom	Å	*degree*	°		

FILES

/usr/pub/eqnchar

SEE ALSO

eqn(1), troff(1).

NAME

fcntl – file control options

SYNOPSIS

```
#include <fcntl.h>
```

DESCRIPTION

The *fcntl*(2) function provides for control over open files. This include file describes *requests* and *arguments* to *fcntl* and *open*(2).

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */
#define O_RDONLY        0
#define O_WRONLY        1
#define O_RDWR  2
#define O_NDELAY        04      /* Non-blocking I/O */
#define O_APPEND        010     /* append (writes guaranteed at the end) */

/* Flag values accessible only to open(2) */
#define O_CREAT 00400   /* open with file create (uses third open arg)*/
#define O_TRUNC 01000   /* open with truncation */
#define O_EXCL  02000   /* exclusive open */

/* fcntl(2) requests */
#define F_DUPFD 0       /* Duplicate fildes */
#define F_GETFD 1       /* Get fildes flags */
#define F_SETFD 2       /* Set fildes flags */
#define F_GETFL 3       /* Get file flags */
#define F_SETFL 4       /* Set file flags */
```

SEE ALSO

fcntl(2), open(2).

NAME

font names – table of font names in short and long formats

SYNOPSIS

cat /usr/lib/fontinfo/kurz

DESCRIPTION

For the usage of fonts other than the default ones in *troff* (or *ltroff* or *vtroff* resp.) the names of these fonts must be specified twice. The full name (see below) is used to control the phototypesetter or the postprocessor (*lcat* or *vcat*). *Troff* itself needs the specification of the font name in a short form for the selection of the corresponding font size tables in a **.fp** -command.

long name	short name	long name	short name
apl	ap	h19	hn
basker.b	bb	hebrew	hb
basker.i	bi	meteor.b	mb
basker.r	br	meteor.i	mi
bocklin	bk	meteor.r	mr
bodoni.b	ob	mona	mn
bodoni.i	oi	nonie.b	nb
bodoni.r	or	nonie.i	ni
chess	ch	nonie.r	nr
clarendon	cl	oldenglish	oe
cm.b	cb	pip	pp
cm.i	ci	playbill	pb
cm.r	cr	script	sc
countdown	co	shadow	sh
cyrillic	cy	sign	sg
delegate.b	db	stare.b	sb
delegate.i	di	stare.i	si
delegate.r	dr	stare.r	sr
fix	fx	times.b	tb
gacham.b	gb	times.i	ti
gacham.i	gi	times.r	tr
gacham.r	gr	times.s	ts
graphics	gf	ugramma	m
greek	gk		

FILES

/usr/lib/fontinfo/kurz

NAME

font list – table of available fonts and point sizes

DESCRIPTION

font	available sizes									
R	6	7	8	9	10	11	12	14	16	18
	20	22	24	28	36					
B	6	7	8	9	10	11	12	14	16	18
	20	22	24	28	36					
I	6	7	8	9	10	11	12	14	16	18
	20	22	24	28	36					
S	6	7	8	9	10	11	12	14	16	18
	20	22	24	28	36					
apl	10									
basker.r	12									
basker.b	12									
basker.i	12									
bocklin	14	28								
bodoni.r	10									
bodoni.b	10									
bodoni.i	10									
chess	18									
clarendon	14	18								
cm.r	6	7	8	9	10	11	12			
cm.b	6	7	8	9	10	11	12			
cm.i	6	7	8	9	10	11	12			
countdown	22									
cyrillic	12									
delegate.r	12									
delegate.b	12									
delegate.i	12									
fix	6	9	10	12	14					
gacham.r	10									
gacham.b	10									
gacham.i	10									
graphics	14									
greek	10									
h19	10									
hebrew	16	17	24	36						
meteor.r	8	10	12							
meteor.b	8	10	12							
meteor.i	8	10								
mona	24									
nonie.r	8	10	12							
nonie.b	8	10	12							
nonie.i	8	10	12							

oldenglish	8	14	18				
pip	16						
playbill	10						
script	18						
shadow	16						
sign	22						
stare.r	8	9	10	11	12	14	16
stare.b	8	9	10	11	12	14	16
stare.i	8	9	10	11	12	14	16
times.r	10						
times.b	10						
times.i	10						
times.s	10						
ugramma	10						

NAME

greek – graphics for the extended TTY-37 type-box

SYNOPSIS

cat /usr/pub/greek [**| greek** –Tterminal]

DESCRIPTION

Greek gives the mapping from ASCII to the "shift-out" graphics in effect between SO and SI on TELETYPE● Model 37 terminals equipped with a 128-character type-box. These are the default greek characters produced by *nroff*(1). The filters of *greek*(1) attempt to print them on various other terminals. The file contains:

lowercase						**uppercase**		
alpha	α	A	omega	ω	C	DELTA	Δ	W
beta	β	B	partial	∂	]	GAMMA	Γ	G
delta	δ	D	phi	φ	U	LAMBDA	Λ	E
epsilon	ε	S	pi	π	J	OMEGA	Ω	Z
eta	η	N	psi	ψ	V	PHI	Φ	F
gamma	γ	\	rho	ρ	K	PI	Π	P
integral	∫	^	sigma	σ	Y	PSI	Ψ	H
lambda	λ	L	tau	τ	I	SIGMA	Σ	R
mu	μ	M	theta	ϑ	O	THETA	Θ	T
nabla	∇	[	xi	ξ	X			
not	¬	_	zeta	ζ	Q			
nu	ν	@						

FILES

/usr/pub/greek

SEE ALSO

300(1), 4014(1), 450(1), greek(1), hp(1), tc(1), troff(1).

NAME

hier – file system hierarchy

DESCRIPTION

The following outline gives a quick tour through a representative directory hierarchy.

```
/       root
/dev/   devices (4)
        console
                main console, termio(4)
        tty*    terminals, termio(4)
        lbp     laser beam printer lbp(4)
        hk*     disks, hk, hk(4)
        rhk*    raw disks, hk, hk(4)
        ...
/bin/   utility programs, cf /usr/bin/ (1)
        as      assembler
        cc      C compiler executive, cf /lib/c[012]
        ...
/lib/   object libraries and other stuff, cf /usr/lib/
        libc.a  system calls, standard I/O, etc. (2,3,3S)
        libffp.a
                math routines fast floating point (3M)
        libieee.a
                math routines ieee floating point (3M)
        libplot.a
                plotting routines, plot(3X)
        ...
        cpp     c pre-processor
        c[012]
                passes of cc(1)
        ...
/etc/   essential data and dangerous maintenance utilities
        passwd
                password file, passwd(5)
        group   group file, group(5)
        motd    message of the day, login(1)
        mnttab
                mounted file table, mnttab(5)
        gettydefs
                terminal characteristics, gettydefs(5)
        inittab
                list of initial processes, inittab(5)
        getty   part of login, getty(8)
        init    the father of all processes, init(8)
        rc      shell program to bring the system up
        cron    the clock daemon, cron(8)
        mount
                mount(8)
        wall    wall(8)
        ...
```

```
/tmp/
        temporary files, usually on a fast device, cf /usr/tmp/
        e*      used by ed(1)
        ctm*    used by cc(1)
        ...
/usr/   general-pupose directory, usually a mounted file system
        adm/    administrative information
                wtmp    login history, utmp(5)
                pacct   process accounting, acct(8)
/usr    /bin
        utility programs, to keep /bin/ small
        tmp/    temporaries, to keep /tmp/ small
        dict/   word lists, etc.
                words   principal word list, used by look(1)
                spellhist
                        history file for spell(1)
        games/
                bj      blackjack
                hangman
                quiz.k/
                        what quiz(6) knows
                        index   category index
                        africa  countries and capitals
                        ...
                ...
        include/
                standard #include files
                a.out.h
                        object file layout, a.out(5)
                stdio.h
                        standard I/O, stdio(3S)
                math.h
                        (3M)
                ...
                sys/    system-defined layouts
                        acct.h  process accounts, acct(5)
                        buf.h   internal system buffers
                        ...
        lib/    object libraries and stuff, to keep /lib/ small
                lint[12]
                        subprocesses for lint(1)
                llib-lc dummy declarations for /lib/libc.a, used by lint(1)
                llib-lm
                        dummy declarations for /lib/libc.m
                atrun   scheduler for at(1)
                struct/
                        passes of struct(1)
                ...
                tmac/
                        macros for troff(1)
                libF77ffp.a
                        Fortran runtime support
```

```
                    libI77ffp.a
                            Fortran I/O
                            tmac.an
                                    macros for man(7)
                            tmac.s
                                    macros for ms(7)
                            ...
                    font/   fonts for troff(1)
                            R       Times Roman
                            B       Times Bold
                            ...
                    uucp/
                            programs and data for uucp(1C)
                            L.sys   remote system names and numbers
                            uucico
                                    the real copy program
                            ...
                    suftab
                            table of suffixes for hyphenation, used by troff(1)
                    units   conversion tables for units(1)
                    eign    list of English words to be ignored by ptx(1)
/usr/   man/
        volume 1 of this manual, man(1)
                    man0/
                            general
                            intro   introduction to volume 1, ms(7) format
                            xx      template for manual page
                    man1/
                            chapter 1
                            as.1
                            mount.1m
                            ...
                    cat1/   preprinted pages for man1/
                            as.1
                            mount.1m
                    ...
        spool/
                    delayed execution files
                    at/     used by at(1)
                    lpd/    used by lpr(1)
                            lock    present when line printer is active
                            cf*     copy of file to be printed, if necessary
                            df*     daemon control file, lpd(8)
                            tf*     transient control file, while lpr is working
                    uucp/
                            work files and staging area for uucp(1C)
                            LOGFILE
                                    summary log
                            LOG.*   log file for one transaction
        mail/       mailboxes for mail(1)
                    uid     mail file for user uid
```

```
            uid.lock
                    lock file while uid is receiving mail
    wd      initial working directory of a user, typically wd is the user's
            login name
            .profile
                    set environment for sh(1), environ(7)
            calendar
                    user's datebook for calendar(1)
    doc/    papers, mostly in volume 2 of this manual, typically in
            ms(7) format
            as/     assembler manual
            c       C manual
            ...
    sys/    unix system configuration
```

SEE ALSO

ls(1), ncheck(8), find(1), grep(1)

BUGS

The position of files is subject to change without notice.

NAME

man – macros for formatting entries in this manual

SYNOPSIS

nroff –man files

troff –man [–rs1] files

DESCRIPTION

These *troff*(1) macros are used to lay out the format of the entries of this manual. A skeleton entry may be found in the file **/usr/man/man0/skeleton**. These macros are used by the *man*(1) command.

The default page size is 8.5"×11", with a 6.5"×10" text area; the **–rs1** option reduces these dimensions to 6"×9" and 4.75"×8.375", respectively; this option (which is *not* effective in *nroff*(1)) also reduces the default type size from 10-point to 9-point, and the vertical line spacing from 12-point to 10-point. The **–rV2** option may be used to set certain parameters to values appropriate for certain Versatec printers: it sets the line length to 82 characters, the page length to 84 lines, and it inhibits underlining; this option should not be confused with the **–Tvp** option of the *man*(1) command, which is available at some UNIX sites.

Any *text* argument below may be one to six "words". Double quotes ("") may be used to include blanks in a "word". If *text* is empty, the special treatment is applied to the next line that contains text to be printed. For example, .I may be used to italicize a whole line, or .SM followed by .B to make small bold text. By default, hyphenation is turned off for *nroff*, but remains on for *troff*.

Type font and size are reset to default values before each paragraph and after processing font- and size-setting macros, e.g., .I, .RB, .SM. Tab stops are neither used nor set by any macro except .DT and .TH.

Default units for indents *in* are ens. When *in* is omitted, the previous indent is used. This remembered indent is set to its default value (7.2 ens in *troff*, 5 ens in *nroff*—this corresponds to 0.5" in the default page size) by .TH, .PP, and .RS, and restored by .RE.

.TH *t s c n*	Set the title and entry heading; *t* is the title, *s* is the section number, *c* is extra commentary, e.g., "local", *n* is new manual name. Invokes .DT (see below).
.SH *text*	Place subhead *text*, e.g., SYNOPSIS, here.
.SS *text*	Place sub-subhead *text*, e.g., Options, here.
.B *text*	Make *text* bold.
.I *text*	Make *text* italic.
.SM *text*	Make *text* 1 point smaller than default point size.
.RI *a b*	Concatenate roman *a* with italic *b*, and alternate these two fonts for up to six arguments. Similar macros alternate between any two of roman, italic, and bold: .IR .RB .BR .IB .BI
.P	Begin a paragraph with normal font, point size, and indent. .PP is a synonym for .P.
.HP *in*	Begin paragraph with hanging indent.
.TP *in*	Begin indented paragraph with hanging tag. The next line that contains text to be printed is taken as the tag. If the tag

	does not fit, it is printed on a separate line.
.IP *t in*	Same as .TP *in* with tag *t*; often used to get an indented paragraph without a tag.
.RS *in*	Increase relative indent (initially zero). Indent all output an extra *in* units from the current left margin.
.RE *k*	Return to the *k*th relative indent level (initially, *k*=1; *k*=0 is equivalent to *k*=1); if *k* is omitted, return to the most recent lower indent level.
.PM *m*	Produces proprietary markings; where *m* may be P for **PRIVATE**, N for **NOTICE**, BP for **BELL LABORATORIES PROPRIETARY**, or BR for **BELL LABORATORIES RESTRICTED**.
.DT	Restore default tab settings (every 7.2 ens in *troff*, 5 ens in *nroff*).
.PD *v*	Set the interparagraph distance to *v* vertical spaces. If *v* is omitted, set the interparagraph distance to the default value (0.4v in *troff*, 1v in *nroff*).

The following *strings* are defined:

*R	® in *troff*(1), (Reg.) in *nroff*(1).
*S	Change to default type size.

The following *number registers* are given default values by .TH:

IN	Left margin indent relative to subheads (default is 7.2 ens in *troff*, 5 ens in *nroff*).
LL	Line length including IN.
PD	Current interparagraph distance.

CAVEATS

In addition to the macros, strings, and number registers mentioned above, there are defined a number of *internal* macros, strings, and number registers. Except for names predefined by *troff*(1) and number registers **d**, **m**, and **y**, all such internal names are of the form *XA*, where *X* is one of),], and }, and *A* stands for any alphanumeric character.

If a manual entry needs to be preprocessed by *cw*(1), *eqn*(1) (or *neqn*), and/or *tbl*(1), it must begin with a special line (described in *man*(1)), causing the *man* command to invoke the appropriate preprocessor(s).

The programs that prepare the Table of Contents and the Permuted Index for this Manual assume the *NAME* section of each entry consists of a single line of input that has the following format:

    name[, name, name ...] \- explanatory text

The macro package increases the inter-word spaces (to eliminate ambiguity) in the *SYNOPSIS* section of each entry.

The macro package itself uses only the roman font (so that one can replace, for example, the bold font by the constant-width font—see *cw*(1)). Of course, if the input text of an entry contains requests for other fonts (e.g., .I, .RB, \fI), the corresponding fonts must be mounted.

FILES

```
/usr/lib/tmac/tmac.an
/usr/lib/macros/cmp.[nt].[dt].an
/usr/lib/macros/ucmp.[nt].an
```

/usr/man/man0/skeleton

SEE ALSO

man(1), nroff(1), troff(1).

BUGS

If the argument to .TH contains *any* blanks and is *not* enclosed by double quotes (""), there will be bird-dropping-like things on the output.

NAME

me − macros for formatting papers

SYNOPSIS

nroff −me [options] file ...
troff −me [options] file ...

DESCRIPTION

This package of *nroff* and *troff* macro definitions provides a canned formatting facility for technical papers in various formats. When producing 2-column output on a terminal, filter the output through *col*(1).

The macro requests are defined below. Many *nroff* and *troff* requests are unsafe in conjunction with this package, however these requests may be used with impunity after the first .pp:

.bp	begin new page
.br	break output line here
.sp n	insert n spacing lines
.ls n	(line spacing) n=1 single, n=2 double space
.na	no alignment of right margin
.ce n	center next n lines
.ul n	underline next n lines
.sz +n	add n to point size

Output of the *eqn*, *neqn*, *refer*, and *tbl*(1) preprocessors for equations and tables is acceptable as input.

FILES

/usr/lib/tmac/tmac.e
/usr/lib/me/*

SEE ALSO

eqn(1), troff(1), refer(1), tbl(1)
−me Reference Manual, Eric P. Allman
Writing Papers with Nroff Using −me

REQUESTS

In the following list, initialization refers to the first .pp, .lp, .ip, .np, .sh, or .uh macro. This list is incomplete; see *The −me Reference Manual* for interesting details.

Request	Initial Value	Cause Break	Explanation
.(c	-	yes	Begin centered block
.(d	-	no	Begin delayed text
.(f	-	no	Begin footnote
.(l	-	yes	Begin list
.(q	-	yes	Begin major quote
.(x *x*	-	no	Begin indexed item in index *x*
.(z	-	no	Begin floating keep
.)c	-	yes	End centered block
.)d	-	yes	End delayed text
.)f	-	yes	End footnote
.)l	-	yes	End list
.)q	-	yes	End major quote
.)x	-	yes	End index item

.)z	-	yes	End floating keep
.++ *m H*	-	no	Define paper section. *m* defines the part of the paper, and can be **C** (chapter), **A** (appendix), **P** (preliminary, e.g., abstract, table of contents, etc.), **B** (bibliography), **RC** (chapters renumbered from page one each chapter), or **RA** (appendix renumbered from page one).
.+c *T*	-	yes	Begin chapter (or appendix, etc., as set by .++). *T* is the chapter title.
.1c	1	yes	One column format on a new page.
.2c	1	yes	Two column format.
.EN	-	yes	Space after equation produced by *eqn* or *neqn*.
.EQ *x y*	-	yes	Precede equation; break out and add space. Equation number is *y*. The optional argument *x* may be *I* to indent equation (default), *L* to left-adjust the equation, or *C* to center the equation.
.TE	-	yes	End table.
.TH	-	yes	End heading section of table.
.TS *x*	-	yes	Begin table; if *x* is *H* table has repeated heading.
.ac *A N*	-	no	Set up for ACM style output. *A* is the Author's name(s), *N* is the total number of pages. Must be given before the first initialization.
.b *x*	no	no	Print *x* in boldface; if no argument switch to boldface.
.ba +*n*	0	yes	Augments the base indent by *n*. This indent is used to set the indent on regular text (like paragraphs).
.bc	no	yes	Begin new column
.bi *x*	no	no	Print *x* in bold italics (nofill only)
.bx *x*	no	no	Print *x* in a box (nofill only).
.ef *'x'y'z'*	''''	no	Set even footer to x y z
.eh *'x'y'z'*	''''	no	Set even header to x y z
.fo *'x'y'z'*	''''	no	Set footer to x y z
.hx	-	no	Supress headers and footers on next page.
.he *'x'y'z'*	''''	no	Set header to x y z
.hl	-	yes	Draw a horizontal line
.i *x*	no	no	Italicize *x*; if *x* missing, italic text follows.
.ip *x y*	no	yes	Start indented paragraph, with hanging tag *x*. Indentation is *y* ens (default 5).
.lp	yes	yes	Start left-blocked paragraph.
.lo	-	no	Read in a file of local macros of the form .**x*. Must be given before initialization.
.np	1	yes	Start numbered paragraph.
.of *'x'y'z'*	''''	no	Set odd footer to x y z
.oh *'x'y'z'*	''''	no	Set odd header to x y z
.pd	-	yes	Print delayed text.
.pp	no	yes	Begin paragraph. First line indented.
.r	yes	no	Roman text follows.
.re	-	no	Reset tabs to default values.
.sc	no	no	Read in a file of special characters and diacritical marks. Must be given before initialization.
.sh *n x*	-	yes	Section head follows, font automatically bold. *n* is level of section, *x* is title of section.
.sk	no	no	Leave the next page blank. Only one page is remembered ahead.

.sz +*n*	10p	no	Augment the point size by *n* points.
.th	no	no	Produce the paper in thesis format. Must be given before initialization.
.tp	no	yes	Begin title page.
.u *x*	-	no	Underline argument (even in *troff*). (Nofill only).
.uh	-	yes	Like .sh but unnumbered.
.xp *x*	-	no	Print index *x*.

NAME

mm − the MM macro package for formatting documents

SYNOPSIS

mm [options] [files]

nroff −mm [options] [files]

nroff −cm [options] [files]

mmt [options] [files]

troff −mm [options] [files]

troff −cm [options] [files]

DESCRIPTION

This package provides a formatting capability for a very wide variety of documents. It is the standard package used by the BTL typing pools and documentation centers. The manner in which a document is typed in and edited is essentially independent of whether the document is to be eventually formatted at a terminal or is to be phototypeset. See the references below for further details.

The **−mm** option causes *nroff*(1) and *troff*(1) to use the non-compacted version of the macro package, while the **−cm** option results in the use of the compacted version, thus speeding up the process of loading the macro package.

FILES

/usr/lib/tmac/tmac.m	pointer to the non-compacted version of the package
/usr/lib/macros/mm[nt]	non-compacted version of the package
/usr/lib/macros/cmp.[nt].[dt].m	compacted version of the package
/usr/lib/macros/ucmp.[nt].m	initializers for the compacted version of the package

SEE ALSO

mm(1), mmt(1), troff(1).
MM—Memorandum Macros by D. W. Smith and J. R. Mashey.
Typing Documents with MM by D. W. Smith and E. M. Piskorik.

NAME

mosd – the OSDD adapter macro package for formatting documents

SYNOPSIS

osdd [options] [files]

mm −mosd [options] [files]

nroff −mm −mosd [options] [files]

nroff −cm −mosd [options] [files]

mmt −mosd [options] [files]

troff −mm −mosd [options] [files]

troff −cm −mosd [options] [files]

DESCRIPTION

The OSDD adapter macro package is a tool used in conjunction with the MM macro package to prepare Operations Systems Deliverable Documentation. Many of the OSDD Standards are different than the default format provided by MM. The OSDD adapter package sets the appropriate MM options for automatic production of the OSDD Standards. The OSDD adapter package also generates the correct OSDD page headers and footers, heading styles, Table of Contents format, etc.

OSDD document (input) files are prepared with the MM macros. Additional information which must be given at the beginning of the document file is specified by the following string definitions:

```
.ds H1 document-number
.ds H2 section-number
.ds H3 issue-number
.ds H4 date
.ds H5 rating
```

The *document-number* should be of the standard 10 character format. The words "Section" and "Issue" should not be included in the string definitions; they will be supplied automatically when the document is printed. For example:

```
.ds H1 OPA−1P135−01
.ds H2 4
.ds H3 2
```

automatically produces

```
OPA-1P135-01
Section 4
Issue 2
```

as the document page header. Quotation marks are not used in string definitions.

If certain information is not to be included in a page header, then the string is defined as null; e.g.,

```
.ds H2
```

means that there is no *section-number*.

The OSDD Standards require that the *Table of Contents* be numbered beginning with *Page 1.* By default, the first page of text will be numbered *Page 2.* If the *Table of Contents* has more than one page, for example *n*, then either **−rP***n+1* must be included as a command line option or **.nr P n** must be included in the document file. For example, if the *Table of Contents* is four pages then use **−rP5** on the command line or **.nr P 4** in the document file.

The OSDD Standards require that certain information such as the document *rating* appear on the *Document Index* or on the *Table of Contents* page if there is no index. By default, it is assumed that an index has been prepared separately. If there is no index, the following must be included in the document file:

```
        .nr Di 0
```

This will ensure that the necessary information is included on the *Table of Contents* page.

The OSDD Standards require that all numbered figures be placed at the end of the document. The **.Fg** macro is used to produce full page figures. This macro produces a blank page with the appropriate header, footer, and figure caption. Insertion of the actual figure on the page is a manual operation. The macro usage is

```
        .Fg page-count "figure caption"
```

where *page-count* is the number of pages required for a multi-page figure (default 1 page).

Figure captions are produced by the **.Fg** macro using the **.BS/.BE** macros. Thus the **.BS/.BE** macros are also not available for users. The **.Fg** macro cannot be used within the document unless the final **.Fg** in a series of figures is followed by a **.SK** macro to force out the last figure page.

The *Table of Contents* for OSDD documents (see Figure 4 in Section 4.1 of the OSDD Standards) is produced with:

```
        .Tc
        System Type
        System Name
        Document Type
        .Td
```

The **.Tc/.Td** macros are used instead of the **.TC** macro from MM.

By default, the adapter package causes the **NOTICE** disclosure statement to be printed. The **.PM** macro may be used to suppress the **NOTICE** or to replace it with the **PRIVATE** disclosure statement as follows:

.PM	none printed
.PM P	PRIVATE printed
.PM N	NOTICE printed (default)

The **.P** macro is used for paragraphs. The **Np** register is set automatically to indicate the paragraph numbering style. It is very important that the **.P** macro be used correctly. All paragraphs (including those immediately following a **.H** macro) must use a **.P** macro. Unless there is a **.P** macro, there will not be a number generated for the paragraph. Similarly, the **.P** macro should not be used for text which is not a paragraph. The **.SP** macro may be appropriate for these cases, e.g., for "paragraphs" within a list item.

The page header format is produced automatically in accordance with the OSDD Standards. The OSDD Adapter macro package uses the **.TP** macro for this purpose. Therefore the **.TP** macro normally available in

MM is not available for users.

FILES

/usr/lib/tmac/tmac.osd

SEE ALSO

mm(1), mmt(1), nroff(1), troff(1), mm(7).
MM—Memorandum Macros by D. W. Smith and J. R. Mashey.
Operations Systems Deliverable Documentation Standards, June 1980.

NAME

mptx – the macro package for formatting a permuted index

SYNOPSIS

nroff –mptx [options] [files]

troff –mptx [options] [files]

DESCRIPTION

This package provides a definition for the **.xx** macro used for formatting a permuted index as produced by *ptx*(1). This package does not provide any other formatting capabilities such as headers and footers. If these or other capabilities are required, the *mptx* macro package may be used in conjuction with the *MM* macro package. In this case, the **–mptx** option must be invoked *after* the **–mm** call. For example:

```
        nroff –cm –mptx file
or
        mm –mptx file
```

FILES

/usr/lib/tmac/tmac.ptx	pointer to the non-compacted version of the package
/usr/lib/macros/ptx	non-compacted version of the package

SEE ALSO

mm(1), nroff(1), ptx(1), troff(1), mm(7).

NAME

ms – macros for formatting manuscripts

SYNOPSIS

nroff –ms [options] file ...
troff –ms [options] file ...

DESCRIPTION

This package of *nroff* and *troff* macro definitions provides a canned formatting facility for technical papers in various formats. When producing 2-column output on a terminal, filter the output through *col*(1).

The macro requests are defined below. Many *nroff* and *troff* requests are unsafe in conjunction with this package, however these requests may be used with impunity after the first .PP:

.bp	begin new page
.br	break output line here
.sp n	insert n spacing lines
.ls n	(line spacing) n=1 single, n=2 double space
.na	no alignment of right margin

Output of the *eqn*, *neqn*, *refer*, and *tbl*(1) preprocessors for equations and tables is acceptable as input.

FILES

/usr/lib/tmac/tmac.s

SEE ALSO

eqn(1), troff(1), refer(1), tbl(1)

REQUESTS

Request	Initial Value	Cause Break	Explanation
.1C	yes	yes	One column format on a new page.
.2C	no	yes	Two column format.
.AB	no	yes	Begin abstract.
.AE	-	yes	End abstract.
.AI	no	yes	Author's institution follows. Suppressed in TM.
.AT	no	yes	Print 'Attached' and turn off line filling.
.AU *x y*	no	yes	Author's name follows. *x* is location and *y* is extension, ignored except in TM.
.B *x*	no	no	Print *x* in boldface; if no argument switch to boldface.
.B1	no	yes	Begin text to be enclosed in a box.
.B2	no	yes	End text to be boxed . print it.
.BT	date	no	Bottom title, automatically invoked at foot of page. May be redefined.
.BX *x*	no	no	Print *x* in a box.
.CS *x*...	-	yes	Cover sheet info if TM format, suppressed otherwise. Arguments are number of text pages, other pages, total pages, figures, tables, references.
.CT	no	yes	Print 'Copies to' and enter no-fill mode.
.DA *x*	nrof	no	'Date line' at bottom of page is *x*. Default is today.
.DE	-	yes	End displayed text. Implies .KE.
.DS *x*	no	yes	Start of displayed text, to appear verbatim line-by-line. *x*=I for indented display (default), *x*=L for left-justified on the page, *x*=C for centered, *x*=B for make left-justified

			block, then center whole block. Implies .KS.
.EG	no	-	Print document in BTL format for 'Engineer's Notes.' Must be first.
.EN	-	yes	Space after equation produced by *eqn* or *neqn*.
.EQ *x y*	-	yes	Precede equation; break out and add space. Equation number is *y*. The optional argument *x* may be *I* to indent equation (default), *L* to left-adjust the equation, or *C* to center the equation.
.FE	-	yes	End footnote.
.FS	no	no	Start footnote. The note will be moved to the bottom of the page.
.HO	-	no	'Bell Laboratories, Holmdel, New Jersey 07733'.
.I *x*	no	no	Italicize *x*; if *x* missing, italic text follows.
.IH	no	no	'Bell Laboratories, Naperville, Illinois 60540'
.IM	no	no	Print document in BTL format for an internal memorandum. Must be first.
.IP *x y*	no	yes	Start indented paragraph, with hanging tag *x*. Indentation is *y* ens (default 5).
.KE	-	yes	End keep. Put kept text on next page if not enough room.
.KF	no	yes	Start floating keep. If the kept text must be moved to the next page, float later text back to this page.
.KS	no	yes	Start keeping following text.
.LG	no	no	Make letters larger.
.LP	yes	yes	Start left-blocked paragraph.
.MF	-	-	Print document in BTL format for 'Memorandum for File.' Must be first.
.MH	-	no	'Bell Laboratories, Murray Hill, New Jersey 07974'.
.MR	-	-	Print document in BTL format for 'Memorandum for Record.' Must be first.
.ND *date*	troff	no	Use date supplied (if any) only in special BTL format positions; omit from page footer.
.NH *n*	-	yes	Same as .SH, with section number supplied automatically. Numbers are multilevel, like 1.2.3, where *n* tells what level is wanted (default is 1).
.NL	yes	no	Make letters normal size.
.OK	-	yes	'Other keywords' for TM cover sheet follow.
.PP	no	yes	Begin paragraph. First line indented.
.PT	pg #	-	Page title, automatically invoked at top of page. May be redefined.
.PY	-	no	'Bell Laboratories, Piscataway, New Jersey 08854'
.QE	-	yes	End quoted (indented and shorter) material.
.QP	-	yes	Begin single paragraph which is indented and shorter.
.QS	-	yes	Begin quoted (indented and shorter) material.
.R	yes	no	Roman text follows.
.RE	-	yes	End relative indent level.
.RP	no	-	Cover sheet and first page for released paper. Must precede other requests.
.RS	-	yes	Start level of relative indentation. Following .IP's are measured from current indentation.
.SG *x*	no	yes	Insert signature(s) of author(s), ignored except in TM. *x* is the reference line (initials of author and typist).

.SH	-	yes	Section head follows, font automatically bold.
.SM	no	no	Make letters smaller.
.TA *x*...	5...	no	Set tabs in ens. Default is 5 10 15 ...
.TE	-	yes	End table.
.TH	-	yes	End heading section of table.
.TL	no	yes	Title follows.
.TM *x*...	no	-	Print document in BTL technical memorandum format. Arguments are TM number, (quoted list of) case number(s), and file number. Must precede other requests.
.TR *x*	-	-	Print in BTL technical report format; report number is *x*. Must be first.
.TS *x*	-	yes	Begin table; if *x* is *H* table has repeated heading.
.UL *x*	-	no	Underline argument (even in troff).
.UX	-	no	'UNIX'; first time used, add footnote 'UNIX is a trademark of Bell Laboratories.'
.WH	-	no	'Bell Laboratories, Whippany, New Jersey 07981'.

NAME

mv – a macro package for making view graphs

SYNOPSIS

mvt [options] [files]
troff –mv [options] [files]

DESCRIPTION

This package provides an easy-to-use facility for making view graphs and projection slides in a variety of formats. A dozen or so macros are provided that accomplish most of the formatting tasks needed in making transparencies. All of the facilities of *troff*(1), *eqn*(1), and *tbl*(1) are available for more difficult tasks. The output can be previewed on most terminals, and, in particular, on the Tektronix 4014 and on the Versatec printer. See the reference below for further details.

FILES

/usr/lib/tmac/tmac.v

SEE ALSO

eqn(1), mvt(1), tbl(1), troff(1).
A Macro Package for View Graphs and Slides by T. A. Dolotta and D. W. Smith (in preparation).

NAME

regexp – regular expression compile and match routines

SYNOPSIS

```
#define INIT <declarations>
#define GETC() <getc code>
#define PEEKC() <peekc code>
#define UNGETC(c) <ungetc code>
#define RETURN(pointer) <return code>
#define ERROR(val) <error code>

#include      <regexp.h>

char *compile(instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;

int step(string, expbuf)
char *string, *expbuf;
```

DESCRIPTION

This page describes general purpose regular expression matching routines in the form of *ed*(1), defined in **/usr/include/regexp.h**. Programs such as *ed*(1), *sed*(1), *grep*(1), *bs*(1), *expr*(1), etc., which perform regular expression matching use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

The interface to this file is unpleasantly complex. Programs that include this file must have the following five macros declared before the "#include <regexp.h>" statement. These macros are used by the *compile* routine.

GETC()	Return the value of the next character in the regular expression pattern. Successive calls to GETC() should return successive characters of the regular expression.
PEEKC()	Return the next character in the regular expression. Successive calls to PEEKC() should return the same character (which should also be the next character returned by GETC()).
UNGETC(*c*)	Cause the argument c to be returned by the next call to GETC() (and PEEKC()). No more that one character of pushback is ever needed and this character is guaranteed to be the last character read by GETC(). The value of the macro UNGETC(c) is always ignored.
RETURN(*pointer*)	This macro is used on normal exit of the *compile* routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.
ERROR(*val*)	This is the abnormal return from the *compile* routine. The argument *val* is an error number (see table below for meanings). This call should never return.

ERROR	MEANING
11	Range endpoint too large.
16	Bad number.
25	"\digit" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\(\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \{ \}.
45	} expected after \.
46	First number exceeds second in \{ \}.
49	[ ] imbalance.
50	Regular expression overflow.

The syntax of the *compile* routine is as follows:

```
compile(instring, expbuf, endbuf, eof)
```

The first parameter *instring* is never used explicitly by the *compile* routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the INIT declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of ((char *) 0) for this parameter.

The next parameter *expbuf* is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter *endbuf* is one more that the highest address that the compiled regular expression may be placed. If the compiled expression cannot fit in (*endbuf−expbuf*) bytes, a call to ERROR(50) is made.

The parameter *eof* is the character which marks the end of the regular expression. For example, in *ed*(1), this character is usually a /.

Each programs that includes this file must have a #define statement for INIT. This definition will be placed right after the declaration for the function *compile* and the opening curly brace ({). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point the beginning of the regular expression so that this register variable can be used in the declarations for GETC(), PEEKC() and UNGETC(). Otherwise it can be used to declare external variables that might be used by GETC(), PEEKC() and UNGETC(). See the example below of the declarations taken from *grep*(1).

There are other functions in this file which perform actual regular expression matching, one of which is the function *step*. The call to *step* is as follows:

```
step(string, expbuf)
```

The first parameter to *step* is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter *expbuf* is the compiled regular expression which was obtained by a call of the function *compile*.

The function *step* returns one, if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable set in *step* is *loc1*. This is a pointer to the first character that matched the regular expression. The variable *loc2*, which is set by the function *advance*, points the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, loc1 will point to the first character of *string* and *loc2* will point to the null at the end of *string*.

Step uses the external variable *circf* which is set by *compile* if the regular expression begins with ^. If this is set then *step* will only try to match the regular expression to the beginning of the string. If more than one regular expression is to be compiled before the the first is executed the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to *step*.

The function *advance* is called from *step* with the same arguments as *step*. The purpose of *step* is to step through the *string* argument and call *advance* until *advance* returns a one indicating a match or until the end of *string* is reached. If one wants to constrain *string* to the beginning of the line in all cases, *step* need not be called, simply call *advance*.

When *advance* encounters a * or \{ \} sequence in the regular expression it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* will back up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at sometime during the backing up process, *advance* will break out of the loop that backs up and will return zero. This is used be *ed*(1) and *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions like **s/y*//g** do not loop forever.

The routines *ecmp* and *getrange* are trivial and are called by the routines previously mentioned.

EXAMPLES

The following is an example of how the regular expression macros and calls look from *grep*(1):

```
#define INIT         register char *sp = instring;
#define GETC()       (*sp++)
#define PEEKC()      (*sp)
#define UNGETC(c)    (--sp)
#define RETURN(c)    return;
#define ERROR(c)     regerr()

#include <regexp.h>
...
              compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
              if(step(linebuf, expbuf))
                        succeed();
```

FILES

/usr/include/regexp.h

SEE ALSO

ed(1), grep(1), sed(1).

BUGS

The handling of *circf* is kludgy.
The routine *ecmp* is equivalent to the Standard I/O routine *strncmp* and should be replaced by that routine.
The actual code is probably easier to understand than this manual page.

NAME

stat – data returned by stat system call

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
```

DESCRIPTION

The system calls *stat* and *fstat*(2) return data whose structure is defined by this include file. The encoding of the field *st_mode* is defined in this file also.

```
struct  stat
{
        dev_t   st_dev;
        ino_t   st_ino;
        unsigned short st_mode;
        short   st_nlink;
        short   st_uid;
        short   st_gid;
        dev_t   st_rdev;
        off_t   st_size;
        time_t  st_atime;
        time_t  st_mtime;
        time_t  st_ctime;
};

#define S_IFMT  0170000         /* type of file */
#define         S_IFDIR 0040000 /* directory */
#define         S_IFCHR 0020000 /* character special */
#define         S_IFBLK 0060000 /* block special */
#define         S_IFREG 0100000 /* regular */
#define         S_IFIFO 0010000 /* fifo */
#define S_ISUID 0004000         /* set user id on execution */
#define S_ISGID 0002000         /* set group id on execution */
#define S_ISVTX 0001000         /* save swapped text even after use */
#define S_IREAD 0000400         /* read permission, owner */
#define S_IWRITE        0000200         /* write permission, owner */
#define S_IEXEC 0000100         /* execute/search permission, owner */
```

FILES

/usr/include/sys/types.h
/usr/include/sys/stat.h

SEE ALSO

stat(2).

NAME

term – conventional names

DESCRIPTION

These names are used by certain commands (e.g., *nroff*(1), *mm*(1), *man*(1), *tabs*(1)) and are maintained as part of the shell environment (see *sh*(1), *profile*(5), and *environ*(7)) in the variable **$TERM**:

1520	Datamedia 1520
1620	Diablo 1620 and others using the HyType II printer
1620–12	same, in 12-pitch mode
2621	Hewlett-Packard HP2621 series
2631	Hewlett-Packard 2631 line printer
2631–c	Hewlett-Packard 2631 line printer - compressed mode
2631–e	Hewlett-Packard 2631 line printer - expanded mode
2640	Hewlett-Packard HP2640 series
2645	Hewlett-Packard HP264n series (other than the 2640 series)
300	DASI/DTC/GSI 300 and others using the HyType I printer
300–12	same, in 12-pitch mode
300s	DASI/DTC/GSI 300s
382	DTC 382
300s–12	same, in 12-pitch mode
3045	Datamedia 3045
33	TELETYPE® Model 33 KSR
37	TELETYPE Model 37 KSR
40–2	TELETYPE Model 40/2
4000A	Trendata 4000A
4014	Tektronix 4014
43	TELETYPE Model 43 KSR
450	DASI 450 (same as Diablo 1620)
450–12	same, in 12-pitch mode
735	Texas Instruments TI735 and TI725
745	Texas Instruments TI745
dumb	generic name for terminals that lack reverse line-feed and other special escape sequences
hp	Hewlett-Packard (same as 2645)
lp	generic name for a line printer
pcsdsg	C-ITOH 101 used in VT52 mode
tn1200	General Electric TermiNet 1200
tn300	General Electric TermiNet 300
tvi970	Televideo 970
vt100	Dec VT100 or compatible

Up to 8 characters, chosen from [–a–z0–9], make up a basic terminal name. Terminal sub-models and operational modes are distinguished by suffixes beginning with a –. Names should generally be based on original vendors, rather than local distributors. A terminal acquired from one vendor should not have more than one distinct basic name.

Commands whose behavior depends on the type of terminal should accept arguments of the form –T*term* where *term* is one of the names given above; if no such argument is present, such commands should obtain the terminal type from the environment variable **$TERM**, which, in turn, should contain *term*.

SEE ALSO

mm(1), nroff(1), tplot(1G), sh(1), stty(1), tabs(1), profile(5), environ(7).

BUGS

This is a small candle trying to illuminate a large, dark problem. Programs that ought to adhere to this nomenclature do so somewhat fitfully.

NAME

types − primitive system data types

SYNOPSIS

```
#include <sys/types.h>
```

DESCRIPTION

The data types defined in the include file are used in UNIX System code; some data of these types are accessible to user code:

```
typedef struct { int r[1]; } * physadr;
typedef struct { long l[1]; } * lphysadr;
typedef long                   daddr_t;
typedef char *                 caddr_t;
typedef unsigned int           uint;
typedef unsigned short         ushort;
typedef ushort                 ino_t;
typedef short                  cnt_t;
typedef long                   time_t;
typedef long                   label_t[9];
typedef short                  dev_t;
typedef long                   off_t;
typedef long                   paddr_t;
typedef long                   key_t;
```

The form *daddr_t* is used for disk addresses except in an i-node on disk, see *fs*(5). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The *label_t* variables are used to save the processor state while another process is running.

SEE ALSO

fs(5).

NAME

intro – introduction to system maintenance procedures

DESCRIPTION

This section outlines certain procedures that will be of interest to those charged with the task of system maintenance. Included are discussions on such topics as boot procedures, recovery from crashes, file backups, etc.

BUGS

No manual can take the place of good, solid experience.

NAME

acctdisk, acctdusg, accton, acctwtmp – overview of accounting and miscellaneous accounting commands

SYNOPSIS

/usr/lib/acct/acctdisk

/usr/lib/acct/acctdusg [**−u** file] [**−p** file]

/usr/lib/acct/accton [file]

/usr/lib/acct/acctwtmp "reason"

DESCRIPTION

Accounting software is structured as a set of tools (consisting of both C programs and shell procedures) that can be used to build accounting systems. *Acctsh*(8) describes the set of shell procedures built on top of the C programs.

Connect time accounting is handled by various programs that write records into **/usr/adm/utmp**, as described in *utmp*(5). The programs described in *acctcon*(8) convert this file into session and charging records, which are then summarized by *acctmerg*(8).

Process accounting is performed by the UNIX System kernel. Upon termination of a process, one record per process is written to a file (normally **/usr/adm/pacct**). The programs in *acctprc*(8) summarize this data for charging purposes; *acctcms*(8) is used to summarize command usage. Current process data may be examined using *acctcom*(1).

Process accounting and connect time accounting (or any accounting records in the format described in *acct*(5)) can be merged and summarized into total accounting records by *acctmerg* (see **tacct** format in *acct*(5)). *Prtacct* (see *acctsh*(8)) is used to format any or all accounting records.

Acctdisk reads lines that contain user ID, login name, and number of disk blocks and converts them to total accounting records that can be merged with other accounting records.

Acctdusg reads its standard input (usually from **find / −print**) and computes disk resource consumption (including indirect blocks) by login. If **−u** is given, records consisting of those file names for which *acctdusg* charges no one are placed in *file* (a potential source for finding users trying to avoid disk charges). If **−p** is given, *file* is the name of the password file. This option is not needed if the password file is **/etc/passwd**.

Accton alone turns process accounting off. If *file* is given, it must be the name of an existing file, to which the kernel appends process accounting records (see *acct*(2) and *acct*(5)).

Acctwtmp writes a *utmp*(5) record to its standard output. The record contains the current time and a string of characters that describe the *reason*. A record type of ACCOUNTING is assigned (see *utmp*(5)). *Reason* must be a string of 11 or less characters, numbers, **$**, or spaces. For example, the following are suggestions for use in reboot and shutdown procedures, respectively:

```
acctwtmp `uname` >> /etc/wtmp
acctwtmp "file save" >> /etc/wtmp
```

FILES

/etc/passwd	used for login name to user ID conversions
/usr/lib/acct	holds all accounting commands listed in sub-class 8 of this manual
/usr/adm/pacct	current process accounting file
/etc/wtmp	login/logoff history file

SEE ALSO

acctcms(8), acctcom(1), acctcon(8), acctmerg(8), acctprc(8), acctsh(8), fwtmp(8), runacct(8), acct(2), acct(5), utmp(5).
UNIX System Accounting

NAME

acctcms – command summary from per-process accounting records

SYNOPSIS

/usr/lib/acct/acctcms [options] files

DESCRIPTION

Acctcms reads one or more *files*, normally in the form described in *acct*(5). It adds all records for processes that executed identically-named commands, sorts them, and writes them to the standard output, normally using an internal summary format. The *options* are:

−a	Print output in ASCII rather than in the internal summary format. The output includes command name, number of times executed, total kcore-minutes, total CPU minutes, total real minutes, mean size (in K), mean CPU minutes per invocation, and "hog factor", as in *acctcom*(1). Output is normally sorted by total kcore-minutes.
−c	Sort by total CPU time, rather than total kcore-minutes.
−j	Combine all commands invoked only once under "***other".
−n	Sort by number of command invocations.
−s	Any file names encountered hereafter are already in internal summary format.

A typical sequence for performing daily command accounting and for maintaining a running total is:

```
acctcms file ... >today
cp total previoustotal
acctcms −s today previoustotal >total
acctcms −a −s today
```

SEE ALSO

acct(8), acctcom(1), acctcon(8), acctmerg(8), acctprc(8), acctsh(8), fwtmp(8), runacct(8), acct(2), acct(5), utmp(5).

NAME

acctcon1, acctcon2 – connect-time accounting

SYNOPSIS

/usr/lib/acct/acctcon1 [options]

/usr/lib/acct/acctcon2

DESCRIPTION

Acctcon1 converts a sequence of login/logoff records read from its standard input to a sequence of records, one per login session. Its input should normally be redirected from **/etc/wtmp**. Its output is ASCII, giving device, user ID, login name, prime connect time (seconds), non-prime connect time (seconds), session starting time (numeric), and starting date and time. The *options* are:

–p Print input only, showing line name, login name, and time (in both numeric and date/time formats).

–t *Acctcon1* maintains a list of lines on which users are logged in. When it reaches the end of its input, it emits a session record for each line that still appears to be active. It normally assumes that its input is a current file, so that it uses the current time as the ending time for each session still in progress. The **–t** flag causes it to use, instead, the last time found in its input, thus assuring reasonable and repeatable numbers for non-current files.

–l *file* *File* is created to contain a summary of line usage showing line name, number of minutes used, percentage of total elapsed time used, number of sessions charged, number of logins, and number of logoffs. This file helps track line usage, identify bad lines, and find software and hardware oddities. Hang-up, termination of *login*(1) and terminiation of the login shell generate a logoff records, so that the number of logoffs is often three to four times the number of sessions. See *init*(8) and *utmp*(5).

–o *file* *File* is filled with an overall record for the accounting period, giving starting time, ending time, number of reboots, and number of date changes.

Acctcon2 expects as input a sequence of login session records and converts them into total accounting records (see **tacct** format in *acct*(5)).

EXAMPLES

These commands are typically used as shown below. The file **ctmp** is created only for the use of *acctprc*(8) commands:

```
acctcon1 –t –l lineuse –o reboots <wtmp | sort +1n +2 >ctmp
acctcon2 <ctmp | acctmerg >ctacct
```

FILES

/etc/wtmp

SEE ALSO

acct(8), acctcms(8), acctcom(1), acctmerg(8), acctprc(8), acctsh(8), fwtmp(8), runacct(8), acct(2), acct(5), utmp(5).

BUGS

The line usage report is confused by date changes. Use *wtmpfix* (see *fwtmp*(8)) to correct this situation.

NAME

acctmerg – merge or add total accounting files

SYNOPSIS

/usr/lib/acct/acctmerg [options] [file] . . .

DESCRIPTION

Acctmerg reads its standard input and up to nine additional files, all in the **tacct** format (see *acct*(5)), or an ASCII version thereof. It merges these inputs by adding records whose keys (normally user ID and name) are identical, and expects the inputs to be sorted on those keys. *Options* are:

–a	Produce output in ASCII version of **tacct**.
–i	Input files are in ASCII version of **tacct**.
–p	Print input with no processing.
–t	Produce a single record that totals all input.
–u	Summarize by user ID, rather than user ID and name.
–v	Produce output in verbose ASCII format, with more precise notation for floating point numbers.

The following sequence is useful for making "repairs" to any file kept in this format:

```
acctmerg –v <file1 >file2
        edit file2 as desired ...
acctmerg –a <file2 >file1
```

SEE ALSO

acct(8), acctcms(8), acctcom(1), acctcon(8), acctprc(8), acctsh(8), fwtmp(8), runacct(8), acct(2), acct(5), utmp(5).

NAME

acctprc1, acctprc2 — process accounting

SYNOPSIS

/usr/lib/acct/acctprc1 [**ctmp**]

/usr/lib/acct/acctprc2

DESCRIPTION

Acctprc1 reads input in the form described by *acct*(5), adds login names corresponding to user IDs, then writes for each process an ASCII line giving user ID, login name, prime CPU time (tics), non-prime CPU time (tics), and mean memory size (in 64-byte units). If **ctmp** is given, it is expected to contain a list of login sessions, in the form described in *acctcon*(8), sorted by user ID and login name. If this file is not supplied, it obtains login names from the password file. The information in **ctmp** helps it distinguish among different login names that share the same user ID.

Acctprc2 reads records in the form written by *acctprc1*, summarizes them by user ID and name, then writes the sorted summaries to the standard output as total accounting records.

These commands are typically used as shown below:

```
acctprc1 ctmp </usr/adm/pacct | acctprc2 >ptacct
```

FILES

/etc/passwd

SEE ALSO

acct(8), acctcms(8), acctcom(1), acctcon(8), acctmerg(8), acctsh(8), fwtmp(8), runacct(8), acct(2), acct(5), utmp(5).

BUGS

Although it is possible to distinguish among login names that share user IDs for commands run normally, it is difficult to do this for those commands run from *cron*(8), for example. More precise conversion can be done by faking login sessions on the console via the *acctwtmp* program in *acct*(8).

NAME

chargefee, ckpacct, dodisk, lastlogin, monacct, nulladm, prctmp, prdaily, prtacct, runacct, shutacct, startup, turnacct – shell procedures for accounting

SYNOPSIS

/usr/lib/acct/chargefee login-name number

/usr/lib/acct/ckpacct [blocks]

/usr/lib/acct/dodisk

/usr/lib/acct/lastlogin

/usr/lib/acct/monacct number

/usr/lib/acct/nulladm file

/usr/lib/acct/prctmp

/usr/lib/acct/prdaily [mmdd]

/usr/lib/acct/prtacct file ["heading"]

/usr/lib/acct/runacct [mmdd] [mmdd state]

/usr/lib/acct/shutacct ["reason"]

/usr/lib/acct/startup

/usr/lib/acct/turnacct on | off | switch

DESCRIPTION

Chargefee can be invoked to charge a *number* of units to *login-name*. A record is written to **/usr/adm/fee**, to be merged with other accounting records during the night.

Ckpacct should be initiated via *cron*(8). It periodically checks the size of **/usr/adm/pacct**. If the size exceeds *blocks*, 1000 by default, *turnacct* will be invoked with argument *switch*. If the number of free disk blocks in the **/usr** file system falls below 500, *ckpacct* will automatically turn off the collection of process accounting records via the **off** argument to *turnacct*. When at least this number of blocks is restored, the accounting will be activated again. This feature is sensitive to the frequency at which *ckpacct* is executed, usually by *cron*.

Dodisk should be invoked by *cron* to perform the disk accounting functions.

Lastlogin is invoked by *runacct* to update **/usr/adm/acct/sum/loginlog**, which shows the last date on which each person logged in.

Monacct should be invoked once each month or each accounting period. *Number* indicates which month or period it is. If *number* is not given, it defaults to the current month (01–12). This default is useful if *monacct* is to executed via *cron*(8) on the first day of each month. *Monacct* creates summary files in **/usr/adm/acct/fiscal** and restarts summary files in **/usr/adm/acct/sum**.

Nulladm creates *file* with mode 664 and insures owner and group are **adm**. It is called by various accounting shell procedures.

Prctmp can be used to print the session record file (normally **/usr/adm/acct/nite/ctmp** created by *acctcon1* (see *acctcon*(8)).

Prdaily is invoked by *runacct* to format a report of the previous day's accounting data. The report resides in **/usr/adm/acct/sum/rprt*mmdd*** where *mmdd* is the month and day of the report. The current daily accounting reports may be printed by typing *prdaily*. Previous days' accounting reports can be printed by using the *mmdd* option and specifying the exact report date desired. Previous daily reports are cleaned up and therefore inaccessible after each invocation of *monacct*.

Prtacct can be used to format and print any total accounting (**tacct**) file.

Runacct performs the accumulation of connect, process, fee, and disk accounting on a daily basis. It also creates summaries of command usage. For more information, see *runacct*(8).

Shutacct should be invoked during a system shutdown (usually in **/etc/shutdown**) to turn process accounting off and append a "reason" record to **/etc/wtmp**.

Startup should be called by **/etc/rc** to turn the accounting on whenever the system is brought up.

Turnacct is an interface to *accton* (see *acct*(8)) to turn process accounting **on** or **off**. The **switch** argument turns accounting off, moves the current **/usr/adm/pacct** to the next free name in **/usr/adm/pacct*incr*** (where *incr* is a number starting with **1** and incrementing by one for each additional **pacct** file), then turns accounting back on again. This procedure is called by *ckpacct* and thus can be taken care of by the *cron* and used to keep **pacct** to a reasonable size.

FILES

/usr/adm/fee	accumulator for fees
/usr/adm/pacct	current file for per-process accounting
/usr/adm/pacct*	used if pacct gets large and during execution of daily accounting procedure
/etc/wtmp	login/logoff summary
/usr/adm/acct/nite	working directory
/usr/lib/acct	holds all accounting commands listed in sub-class 8 of this manual
/usr/adm/acct/sum	summary directory, should be saved

SEE ALSO

acct(8), acctcms(8), acctcom(1), acctcon(8), acctmerg(8), acctprc(8), fwtmp(8), runacct(8), acct(2), acct(5), utmp(5).

NAME

bcopy – interactive block copy

SYNOPSIS

/etc/bcopy

DESCRIPTION

Bcopy dates from a time when neither the UNIX System file nor the DEC disk drives were as reliable as they are now. *Bcopy* copies from and to files starting at arbitrary block (512-byte) boundaries.

The following questions are asked:

to:	(you name the file or device to be copied to).
offset:	(you provide the starting "to" block number).
from:	(you name the file or device to be copied from).
offset:	(you provide the starting "from" block number).
count:	(you reply with the number of blocks to be copied).

After **count** is exhausted, the **from** question is repeated (giving you a chance to concatenate blocks at the **to+offset+count** location). If you answer **from** with a carriage return, everything starts over.

Two consecutive carriage returns terminate *bcopy*.

SEE ALSO

cpio(1), dd(1).

NAME

boot − standalone startup program

DESCRIPTION

The Cadmus "minitor" contains drivers for a few disks or tapes only. If you want to boot a program from another device, you can boot the program "boot" first, which includes many more drivers than the minitor, and use "boot" to boot the desired program from the desired device. To load boot, the user types on the system console the string "/sa1/boot", if the programs are to be loaded from a 512 byte device, or "/sa2/boot", if the device is a 1kb device, followed by a carriage return; the named program is retrieved from the file system that starts at block 0 of drive 0 of the boot device.

When executed (type g after loading), *boot* sets up memory management, relocates itself into high memory, and types a ':' on the console. Then it reads from the console a device specification (see below) followed immediately by a pathname. *Boot* finds the corresponding file on the given device, loads that file into memory location zero, sets up memory management as required, and calls the program by jumping to location 0. Normal line editing characters can be used.

Conventionally, the name of the boot program is '/sa[12]/boot' and the name of the current version of the system is '/unix'. Then, the recipe is:

1) Specify to the minitor the device, on which boot resides, if necessary, by typing "rl" or "rw" or "rs" etc.

2) Type */sa1/boot*resp*/sa2/boot*.

3) When the prompt is given, type e.g.

        hk(0,0)unix

    or

        st(0,5)check

    depending on whether you are loading from an HK or a streamer respectively. The first 0 indicates the physical unit number; the second indicates the block number of the beginning of the logical file system to be searched, resp. the file number on the tape. (See below).

Device specifications. A device specification has the following form:

    device(unit,offset)

where *device* is the type of the device to be searched, *unit* is the unit number of the device, and *offset* is the block offset of the file system on a disk, or the file number on a tape. *Device* is one of the following

rm	RM02/03
rl	RL01/02
hk	RK06/07
rp	RP03
hp	RP04/5/6
rk	RK05
rx	RX01/2
tm	TM16
ht	TE10
ot	5 1/4 Win
td	5 1/4 Win
st	Streamer

For example, the specification

hp(1,7000)

indicates an RP03 disk, unit 1, and the file system found starting at block 7000 (cylinder 35).

It is seldom necessary to use boot, as the programs can often be loaded directly with the minitor. Boot however must be used to load programs from a tape.

FILES

/unix – system code
/sa1 – directory with standalone programs for 512 byte Filesystems
/sa2 – directory with standalone programs for 1024 byte Filesystems
/sa[12]/boot – bootstrap
/sa[12]/mkfs – mkfs

SEE ALSO

standalone(8)

NAME

brc, bcheckrc, rc, powerfail – system initialization shell scripts

SYNOPSIS

/etc/brc

/etc/bcheckrc

/etc/rc

/etc/powerfail

DESCRIPTION

Except for *powerfail*, these shell procedures are executed via entries in */etc/inittab* by *init*(8) when the system is changed out of *SINGLE USER* mode. *Powerfail* is executed whenever a system power failure is detected.

The *brc* procedure clears the mounted file system table, */etc/mnttab* (see *mnttab*(5)), and loads any programmable micro-processors with their appropriate scripts.

The *bcheckrc* procedure performs all the necessary consistency checks to prepare the system to change into multi-user mode. It will prompt to set the system date and to check the file systems with *fsck*(8).

The *rc* procedure starts all system daemons before the terminal lines are enabled for multi-user mode. In addition, file systems are mounted and accounting, error logging, system activity logging and the Remote Job Entry (RJE) system are activated in this procedure.

The *powerfail* procedure is invoked when the system detects a power failure condition. Its chief duty is to reload any programmable micro-processors with their appropriate scripts, if appropriate. It also logs the fact that a power failure occurred.

These shell procedures, in particular *rc* may be used for several run-level states. The who(1) command may be used to get the run-level information.

BUGS

Powerfail signals are not yet supported on **CADMUS**.

SEE ALSO

init(8), shutdown(8), who(1), inittab(5).

NAME

catman – create the cat files for the manual

SYNOPSIS

/usr/ucb/catman [**–p**] [**–n**] [**–w**] [sections]

DESCRIPTION

Catman creates the preformatted versions of the on-line manual from the nroff input files. Each manual page is examined and those whose preformatted versions are missing or out of date are recreated. If any changes are made, *catman* will recreate the **/usr/lib/whatis** database.

If there is one parameter not starting with a '–', it is take to be a list of manual sections to look in. For example

catman 123

will cause the updating to only happen to manual sections 1, 2, and 3.

Options:

–n	prevents creations of **/usr/lib/whatis**.
–p	prints what would be done instead of doing it.
–w	causes only the **/usr/lib/whatis** database to be created. No manual reformatting is done.

FILES

/usr/man/man?/*.*	raw (nroff input) manual sections
/usr/man/cat?/*.*	preformatted manual pages
/usr/lib/makewhatis	commands to make whatis database

SEE ALSO

man(1), whatis(1)

NAME

check – disk checking and formatting

SYNOPSIS

/sa1/check

/sa2/check

DESCRIPTION

Check is the *Cadmus* disk checking program. Additionally it has a formatting capability for disks with standard headers. *Check* tests disks for the location of bad sectors and writes the *bad sector file* onto disks. The bad sector file is a list of all bad sectors found on a disk. *MUNIX* uses this information to avoid allocating bad sectors to a user's file. If there is an error in a header, or if there is a read or write error within one sector, that sector is defined as a *bad sector*. If possible the header of this sector is marked.

The devices in the following table are supported by *check*. Devices indicated by *YES* in the column *Formatting* uses standard headers and can be formatted by *check*. For other devices exists a specific standalone formatting program (i.e. *rlformat*).

Supported Devices

Device Name	Disk Type	CSR Address	Formatting
hk	RK06/07	FFFF20	YES
rl	RL01/02	FFF900	NO
hl	RL01/02	FFF910	NO
rm	RM02/03/05	FFFDC0	YES

Check is a standalone program. Load and start it by the *Minitor* (see *Minitor*-Manual). For example type:

.rl	(load from RL02)
./sa[12]/check	(executable file)
.g0	(start the program)

You will get a list of all supported devices (i.e. *rl hl hk rm*). Type the device name and the unit number of the disk to be tested or formatted. The input format for opening a device is as follows:

devname(*unit*) [-r] [-p] | exit

where *devname* is one of the device names from the table above and *unit* is the number of the physical drive to be tested. The option **-p** opens the disk in *preserve-mode*, while **-r** opens the disk in *read-only-mode*. A missing option opens the disk in *read/write-mode*.
Exit stops execution of the check program.

In *read/write-mode* the contents of the disk is overwritten, bad sectors are marked, the bad sector file is initialized or modified and formatting is possible. This is the proper mode for new disks.
In *preserve-mode* the contents of the disk is left unchanged. Sectors are tested only by reading, no formatting is done, no sector is marked as bad and the contents of an existing bad sector file is not modified.
In *read-only-mode* sectors ar tested only by reading, bad sectors are

marked and the bad sector file is initialized or modified. Formatting is inhibited.
For example (user input is **bold**):

```
type: devname(unit) [-r] [-p] | exit
: rl(0) -r or
: hk(4) or
: exit
```

If you entered a legal device name you are now in command mode. You will get a list of available commands. Every command you choose refers to the previously specified device and unit. To leave the command mode simply type **q**. The other commands are explained below. The command descriptions refer to disks opened in *read/write-mode*. If you opened a disk in another mode read the descriptions accordingly:

b *Bad Sector Scan:*
The complete disk is tested. Sectors are first written in increasing order and then read in decreasing order. A bad sector file is written onto the disk.

s *Selected Sector Test:*
Consecutive sectors are tested. Choose the starting sector and the number of sectors to be tested. The sectors are tested alternately: *1st sector*, *last sector*, *2nd sector*, ... to the midst of the given interval. The already existing bad sector file is updated.

a *Append Bad Sectors:*
Bad sectors are appended manually to an existing bad sector file. Type in the numbers of sectors to be marked as bad sectors. Type −1 to exit from this command.
This command is extremely helpful if you know any bad sectors not detected by the check program.

i *Inspect Bad Sector File:*
List the contents of an existing bad sector file.

r *Random Sector Test:*
Test disk sectors in random order. Exit from this command by pushing INIT. An existing bad sector file is destroyed.

f *Format Disk:*
Write good sector headers and initialize data fields optionally on the complete disk volume or on single tracks.

? *List Commands:*
Print a table of all available commands.

As a proper test strategy for new disks we suggest command **b** (very fast) followed by command **s** with the whole disk as the sector interval (very slow: you better go for lunch). If there is a new bad sector on an already used disk test a small range around the bad sector by command **s** or use command **a** to mark the bad sector manually. Used disks should be checked in *read-only-mode*.

Each command can be interrupted by pushing INIT. Then you will get the *Minitor* prompt. To restart *check* you have to type **g0**.

SEE ALSO
rl(4), rm(4), hk(4), iopage(7), format(8), standalone(8)
Bad Sector Handling (Vol. 2c)

Minitor-Manual

NAME

checkall — faster file system checking procedure

SYNOPSIS

/etc/checkall

DESCRIPTION

The *checkall* procedure is a prototype and must be modified to suit local conditions. The following will serve as a example:

```
# check the root file system by itself
fsck /dev/hk0

# dual fsck of drives 0 and 1
dfsck /dev/rhk[123] — /dev/rhk[456]
```

Dfsck is a program that permits an operator to interact with two *fsck*(8) programs at once. To aid in this, *dfsck* will print the file system name for each message to the operator. When answering a question from *dfsck*, the operator must prefix the response with a **1** or a **2** (indicating that the answer refers to the first or second file system group).

Due to the file system load balancing required for dual checking, the *dfsck* command should always be executed through the *checkall* shell procedure.

In a practical sense, the file systems are divided up as follows:

```
dfsck file_systems_on_drive_0 — file_systems_on_drive_1
dfsck file_systems_on_drive_2 — file_systems_on_drive_3
   . . .
```

A three drive system can be handled by this more concrete example (assumes two large file systems per drive):

```
dfsck /dev/dsk31 /dev/dsk[14] — /dev/dsk1[14] /dev/dsk34
```

Note that the first drive 3 file system is first in the *filesystems1* list and is last in the *filesystems2* list assuring that references to that drive will not overlap at execution time.

WARNINGS

1. Do not use *dfsck* to check the *root* file system.
2. On a check that requires a scratch file (see **—t** above), be careful not to use the same temporary file for the two groups (this is sure to scramble the file systems).
3. The *dfsck* procedure is useful only if the system is set up for multiple physical I/O buffers.

SEE ALSO

fsck(8).

NAME

chroot – change root directory for a command

SYNOPSIS

/etc/chroot newroot command

DESCRIPTION

The given command is executed *relative to the new root*. The meaning of any initial slashes (/) in path names is changed for a command and any of its children to *newroot*. Furthermore, the initial working directory is *newroot*.

Notice that:

```
chroot newroot command >x
```

will create the file **x** relative to the original root, not the new one.

This command is restricted to the super-user.

The new root path name is always relative to the current root: even if a *chroot* is currently in effect, the *newroot* argument is relative to the current root of the running process.

SEE ALSO

chdir(2).

BUGS

One should exercise extreme caution when referencing special files in the new root file system.

NAME

clri – clear i-node

SYNOPSIS

/etc/clri file-system i-number ...

DESCRIPTION

Clri writes zeros on the 64 bytes occupied by the i-node numbered *i-number*. *File-system* must be a special file name referring to a device containing a file system. After *clri* is executed, any blocks in the affected file will show up as "missing" in an *fsck*(8) of the *file-system*. This command should only be used in emergencies and extreme care should be exercised.

Read and write permission is required on the specified *file-system* device. The i-node becomes allocatable.

The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to *zap* an i-node which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the i-node is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated i-node, so the whole cycle is likely to be repeated again and again.

SEE ALSO

fsck(8), fsdb(8), ncheck(8), fs(5).

BUGS

If the file is open, *clri* is likely to be ineffective.

NAME

crash - what to do when the system crashes

DESCRIPTION

System crashes are of course not something we like to discuss. But if you suffer from crashes, you may appreciate the following hints.

Scenario 1

The system hangs. Users who are not in an editor still get characters echoed, but no program terminates. The shell may even continue to prompt, but when the user gives a command, it hangs too. This normally indicates the loss of a disk interrupt. If only one terminal blocks, this indicates the loss of a terminal interrupt. See "Q-Bus order" below. What to do: push the HALT button, type "n" if the system asks you "Continue? (y,n)". This at least flushes the buffer pool onto the disk and you will have less corrupted files when you reboot.

Scenario 2

The system hangs. No terminal echoes. The console shows no crash message. The RUN light flickers or is off. Pushing the HALT button shows no effect. This indicates a double bus error, which stopped the cpu. Possible reason: memory error, wrong dma registers, see below. What to do: nothing. Reboot.

Scenario 3

The system crashes, and the console shows a nice crash message. The message starts with a string "panic: ...". The most important strings are:

1) panic: Timeout table overflow. Increase the constant NCALL in /usr/sys/conf.h and generate a new unix.

2) panic: iinit. Could not read the super block of the root file system. Probably ROOTDEV in /usr/sys/conf.h is wrong, i.e. the system tries to access the wrong disk.

3) panic: out of swap space. Increase the swap space, i.e. decrease SWPLO and/or increase NSWAP in /usr/sys/conf.h.

Scenario 4

The system crashes, the console shows a crash message, but no "panic: ...". First, look at the cause of the crash, e.g. "bus error", "address error", "illegal vector interrupt", etc. .

1) Bus or address error: if the access address is in the I/O page, then the kernel tried to access a non-existent device, or the device malfunctioned. From the access address try to locate the device (see *confinfo*(4)).

   If the access address is not in the I/O page, but "makes sense", i.e. is not longer than 24 bits and refers to system or user code or data, then perhaps the memory has an error. Lets assume the access address is 0x3000fe. The first 4 bit are a 3, so the address refers to segment 3, offset 0xfe. The crash message shows the mmu settings, i.e. the physical start of each segment. Lets assume mmu[3] is set to 0x40200. Then the physical address corresponding to 0x3000fe is 0x40200 + 0xfe = 0x402fe. If another crash has access address 0x4002fe and mmu[4] = 0x40000, then again the physical address is 0x402fe. So there may be a defective memory chip. Run the program /sa1/memtst to check the memory.

If the access address does not make sense, e.g. is 0xffffffff, or 0x3c004ab9, then it gets more difficult. Look at the more general sections below.

2) illegal vector interrupt: There is a device which interrupts to a vector that has not been generated. Vectors are generated in /usr/sys/l.s. First try to find the interrupting device. Normally the message comes "always when I write to /dev/...". Next, why does it jump to the wrong vector? Check the standard vector in /usr/sys/l.s with the switches on the controller board. If you think you know where the interrupt actually goes to, make sure. Lets assume you think the vector goes to 0x1f4, but you are not sure. Boot unix, but before the minitor "g", write an odd number, e.g. 7, into 0x1f4: 1f4.4 = 7. Then start the system. If you get now instead of "illegal vector interrupt" the message "address error, access address = 7", then you know you were right.

3) exception 24: You have probably a hole in the daisy chain of the Q-Bus. The exception 24 is the so called spurious interrupt. This is taken when a bus timeout occurs while the cpu wants to read an interrupt vector. Most often it indicates an interrupted daisy chain. All cards in the Q-Bus must be on a zig-zag line in the card cage, with no holes in between.

4) exception 30: This is the "HALT" exception. It occurs when you push the HALT button, but may also occur when a DLV11 still has the HALT or INIT jumper and someone pushes the BREAK key on the keyboard or switches the terminal off. Remove the jumper(s).

5) zero divide: You overlooked the foregoing "panic: ..." message. See above.

The crash message

If you have not yet seen a crash message, you are lucky. Don't push your luck by reading further. Otherwise, if you want to see one just for learning, go into single user mode, enter "sync" and push the HALT button. The console will display about half a screenful of lines and ask "Continue? (y,n)". Enter "y".

The message starts with the contents of the registers. Then follows the cause of the crash, e.g. "exception 30" if you pushed the HALT button. Then come the processor status register and the program counter.

After the state at the time of crash comes a small procedure backtrace. This is headed by the line "procedure addresses:lineno". This means that the backtrace is written as a pair of numbers, where the first number is the address of the first instruction of the procedure, and the second the line number the procedure is currently executing. You can always ignore the second number, as your code is not compiled with the -L option of the C compiler. It remains the first number. You can identify this address in the file /usr/sys/unix.sym, which contains the addresses of all global references in the kernel, sorted by name and by value. Or you enter the command "adb /unix /dev/kmem" and enter the address followed by ?i, e.g. #3fa2?i. You will see something like "_sleep: link a6,@-26", so you know 0x3fa2 is the address of the routine sleep. Some of the numbers in the backtrace can normally not be identified. They generally have the format xxxx4e9x. This procedure was called by an indirect jsr, e.g. jsr (a4).

The service department can do nothing, when called by telephone or sent mail, with the pure numbers of the stack trace, as they are different from machine to machine, but gets much better information when you can give the names of the active procedures.

General causes of crashes

Wrong DMA generation

The most frequent error is the wrong choice of 18 or 22 bit drivers for certain controllers, or the wrong choice for DMA extension registers (DER). This error will often not become apparent until either someone accesses the raw device, or the system starts swapping. In the normal case, transfer from disk goes only to system buffers in the first 256 kbytes of memory, so that an 18 bit controller has no problems. But during swapping or raw device I/O, DMA may access any memory address. If your system is only lightly loaded, you read from a raw device, and immediately the system stops, often not even giving an explanation, then most probably the DMA transfer took place to a wrong 256k segment in memory, often the first 256k, overwriting the system code!

Q-Bus order

Contrary to popular belief, the order of cards on the Q-Bus is not immaterial. The first Q-Bus specification allowed only level 4 interrupts, and there are still cards on the market that don't recognize level 5 or level 6 interrupts coming from devices behind them, like old DLV11s. The best rule today is: put higher level cards into the upper slots, lower level cards lower, and level 4 cards at the end. If you suspect a certain controller, put it to the end for a try.

The wrong bus order is sometimes a cause of lost interrupts, or of interrupts coming with the wrong priority. The latter is extremely hard to diagnose, as it screws up internal pointers, which leads to a crash only some time later. But observation helps. If you can state e.g. "Always when I read the tape and at the same time have much output on the DZ, the system is likely to crash", and the DZ is in front of the tape controller, reverse the two.

Static electricity

You may find that your system crashes when air humidity goes down. Don't laugh! Static electricity builds up more in dry rooms. If you can even feel a spark on your finger tip when you touch the system, then you must do something about it. We once installed an air humidifier near a system, but even better is an antistatic rubber mat.

Spikes

Spikes or RFI may leak in via the terminal lines or the power line. We once had the case that the system crashed nearly every time the cleaning lady approached the system with a strong vacuum cleaner! What to do: don't let her come near!

NAME

cron – clock daemon

SYNOPSIS

/etc/cron

DESCRIPTION

Cron executes commands at specified dates and times according to the instructions in the file **/usr/lib/crontab**. Because *cron* never exits, it should be executed only once. This is best done by running *cron* from the initialization process through the file **/etc/rc** (see *init*(8)).

The file **crontab** consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns that specify in order:

minute (0-59),
hour (0-23),
day of the month (1-31),
month of the year (1-12),
and day of the week (0-6, with 0=Sunday).

Each of these patterns may contain:

a number in the (respective) range indicated above;
two numbers separated by a minus (indicating an inclusive range);
a list of numbers separated by commas (meaning all of these numbers); or
an asterisk (meaning all legal values).

NAME

dcopy – copy file systems for optimal access time

SYNOPSIS

/etc/dcopy [**–s**X] [**–a**n] [**–d**] [**–v**] [**–f**fsize:isize] inputfs outputfs

DESCRIPTION

Dcopy copies file system *inputfs* to *outputfs*. *Inputfs* is the existing file system; *outputfs* is an appropriately sized file system, to hold the reorganized result. For best results *inputfs* should be the raw device and *outputfs* should be the block device. *Dcopy* should be run on unmounted file systems (in the case of the root file system, copy to a new pack). With no arguments, *dcopy* copies files from *inputfs* compressing directories by removing vacant entries, and spacing consecutive blocks in a file by the optimal rotational gap. The possible options are

–s*X* supply device information for creating an optimal organization of blocks in a file. The forms of *X* are the same as the **–s** option of *fsck*(8).

–a*n* place the files not accessed in *n* days after the free blocks of the destination file system (default for *n* is 7). If no *n* is specified then no movement occurs.

–d leave order of directory entries as is (default is to move subdirectories to the beginning of directories).

–v currently reports how many files were processed, and how big the source and destination freelists are.

–f*fsize*[:*isize*]
specify the *outputfs* file system and inode list sizes (in blocks). If not given, the values from the *inputfs* are used.

Dcopy catches interrupts and quits and reports on its progress. To terminate *dcopy*, send a quit signal and *dcopy* will no longer catch interrupts or quits. *Dcopy* also attempts to modify its command line arguments so its progress can be monitored with *ps*(1).

SEE ALSO

fsck(8), mkfs(8), ps(1).

NAME

devnm − device name

SYNOPSIS

/etc/devnm [names]

DESCRIPTION

Devnm identifies the special file associated with the mounted file system where the argument *name* resides (as a special case, both the block device name and the swap device name is printed for the argument name **/** if swapping is done on the same disk section as the **root** file system). Argument names must be full path names.

This command is most commonly used by **/etc/rc** (see *bcheckrc*(8)) to construct a mount table entry for the **root** device.

EXAMPLE

The command:

```
/etc/devnm /usr
```

produces

```
rp1 /usr
```

if **/usr** is mounted on **/dev/rp1**.

FILES

/etc/mnttab

SEE ALSO

bcheckrc(8), setmnt(8).

NAME

df − report number of free disk blocks

SYNOPSIS

df [**−t**] [**−f**] [file-systems]

DESCRIPTION

Df prints out the number of free blocks and free i-nodes available for on-line file systems by examining the counts kept in the super-blocks; *file-systems* may be specified either by device name (e.g., **/dev/rl2**) or by mounted directory name (e.g., **/usr**). If the *file-systems* argument is unspecified, the free space on all of the mounted file systems is printed.

The **−t** flag causes the total allocated block figures to be reported as well.

If the **−f** flag is given, only an actual count of the blocks in the free list is made (free i-nodes are not reported). With this option, *df* will report on raw devices.

FILES

/etc/mnttab

SEE ALSO

fs(5), mnttab(5).

NAME

dmesg – collect system diagnostic messages to form error log

SYNOPSIS

/etc/dmesg [–]

DESCRIPTION

Dmesg looks in a system buffer for recently printed diagnostic messages and prints them on the standard output. The messages are those printed by the system when device (hardware) errors occur and (occasionally) when system tables overflow non-fatally. If the – flag is given, then *dmesg* computes (incrementally) the new messages since the last time it was run and places these on the standard output. This is typically used with *cron*(8) to produce the error log */usr/adm/messages* by running the command

```
/etc/dmesg – >> /usr/adm/messages
```

every 10 minutes.

FILES

/usr/adm/messages	error log (conventional location)
/usr/adm/msgbuf	scratch file for memory of – option

BUGS

The system error message buffer is of small finite size. As *dmesg* is run only every few minutes, not all error messages are guaranteed to be logged. This can be construed as a blessing rather than a curse.

Error diagnostics generated immediately before a system crash will never get logged.

NAME

dump – incremental file system dump

SYNOPSIS

dump [key [argument ...] filesystem]

DESCRIPTION

Dump copies to magnetic tape all files changed after a certain date in the *filesystem*. The *key* specifies the date and other options about the dump. *Key* consists of characters from the set **0123456789fusd.**

f Place the dump on the next *argument* file instead of the tape.

u If the dump completes successfully, write the date of the beginning of the dump on file '/etc/ddate'. This file records a separate date for each filesystem and each dump level.

0–9 This number is the 'dump level'. All files modified since the last date stored in the file '/etc/ddate' for the same filesystem at lesser levels will be dumped. If no date is determined by the level, the beginning of time is assumed; thus the option **0** causes the entire filesystem to be dumped.

s The size of the dump tape is specified in feet. The number of feet is taken from the next *argument*. When the specified size is reached, the dump will wait for reels to be changed. The default size is 2300 feet.

d The density of the tape, expressed in BPI, is taken from the next *argument*. This is used in calculating the amount of tape used per write. The default is 1600.

b The size of the output file (most often a floppy) in 512 byte blocks is taken from the next *argument*.

S This option is used to dump onto the streamer. The default output file is /dev/rst0, and the blocking factor is increased.

If no arguments are given, the *key* is assumed to be **9u** and a default file system is dumped to the default tape.

Now a short suggestion on how perform dumps. Start with a full level 0 dump

```
dump 0u
```

Next, periodic level 9 dumps should be made on an exponential progression of tapes. (Sometimes called Tower of Hanoi – 1 2 1 3 1 2 1 4 ... tape 1 used every other time, tape 2 used every fourth, tape 3 used every eighth, etc.)

```
dump 9u
```

When the level 9 incremental approaches a full tape (about 78000 blocks at 1600 BPI blocked 20), a level 1 dump should be made.

```
dump 1u
```

After this, the exponential series should progress as uninterrupted. These level 9 dumps are based on the level 1 dump which is based on the level 0 full dump. This progression of levels of dump can be carried as far as desired.

FILES

default filesystem and tape vary with installation.
/etc/ddate: record dump dates of filesystem/level.
/bin/dump: dump program for 1kbyte filesystems.
/bin/dump.1b: dump program for 512byte filesystems.

SEE ALSO

restor(8), dump(5), dumpdir(8)

DIAGNOSTICS

If the dump requires more than one tape, it will ask you to change tapes. Reply with a new-line when this has been done.

BUGS

Sizes are based on 1600 BPI blocked tape. The raw magtape device has to be used to approach these densities. Read errors on the filesystem are ignored. Write errors on the magtape are usually fatal.

NAME

dumpdir – print the names of files on a dump tape

SYNOPSIS

dumpdir [**f** filename]

DESCRIPTION

Dumpdir is used to read magtapes dumped with the *dump* command and list the names and inode numbers of all the files and directories on the tape.

The **f** option causes *filename* as the name of the tape instead of the default.

FILES

default tape unit varies with installation
rst*

SEE ALSO

dump(8), restor(8)

DIAGNOSTICS

If the dump extends over more than one tape, it may ask you to change tapes. Reply with a new-line when the next tape has been mounted.

BUGS

There is redundant information on the tape that could be used in case of tape reading problems. Unfortunately, *dumpdir* doesn't use it.

NAME

expire − remove outdated news articles

SYNOPSIS

/usr/lib/news/expire [**−n** *newsgroups*] [**−i**] [**−I**] [**−v** [*level*]] [**−e***days*] [**−a**] [**−r**] [**−h**]

DESCRIPTION

Expire is normally started up by *cron*(8) every night to remove all expired news. If no newsgroups are specified, the default is to expire **all**.

Articles whose specified expiration date has already passed are considered expirable. The **−a** option causes expire to archive articles in /usr/spool/oldnews. Otherwise, the articles are unlinked.

The **−v** option causes expire to be more verbose. It can be given a verbosity level (default 1) as in **−v3** for even more output. This is useful if articles aren't being expired and you want to know why.

The **−e** flag gives the number of days to use for a default expiration date. If not given, an installation dependent default (often 2 weeks) is used.

The **−i** and **−I** flags tell *expire* to ignore any expiration date explicitly given on articles. This can be used when disk space is really tight. The **−I** flag will always ignore expiration dates, while the **−i** flag will only ignore the date if ignoring it would expire the article sooner. *WARNING:* If you have articles archived by giving them expiration dates far into the future, these options might remove these files anyway.

The **−r** flag rebuilds the history file without removing any files. In the process, *expire* formats the *dbm*(3X) format files associated with the history file.

The **−h** flag expires articles without using the history file. Both the **−r** and **−h** flags use the active file for newsgroup information rather than the history file.

SEE ALSO

checknews(1), inews(1), readnews(1), recnews(8), sendnews(8), uurec(8)

NAME

ff − list file names and statistics for a file system

SYNOPSIS

/etc/ff [options] special

DESCRIPTION

Ff reads the i-list and directories of the *special* file, assuming it to be a file system, saving i-node data for files which match the selection criteria. Output consists of the path name for each saved i-node, plus any other file information requested using the print *options* below. Output fields are positional. The output is produced in i-node order; fields are separated by tabs. The default line produced by *ff* is:

path-name i-number

With all *options* enabled, output fields would be:

path-name i-number size uid

The argument *n* in the *option* descriptions that follow is used as a decimal integer (optionally signed), where *+n* means more than *n*, *−n* means less than *n*, and *n* means exactly *n*. A day is defined as a 24 hour period.

−I	Do not print the i-node number after each path name.
−l	Generate a supplementary list of all path names for multiply linked files.
−p *prefix*	The specified *prefix* will be added to each generated path name. The default is ..
−s	Print the file size, in bytes, after each path name.
−u	Print the owner's login name after each path name.
−a *n*	Select if the i-node has been accessed in *n* days.
−m *n*	Select if the i-node has been modified in *n* days.
−c *n*	Select if the i-node has been changed in *n* days.
−n *file*	Select if the i-node has been modified more recently than the argument *file*.
−i *i-node-list*	Generate names for only those i-nodes specified in *i-node-list*.

EXAMPLES

To generate a list of the names of all files on a specified file system:

```
ff −I /dev/diskroot
```

To produce an index of files and i-numbers which are on a file system and have been modified in the last 24 hours:

```
ff −m −1 /dev/diskusr > /log/incbackup/usr/tuesday
```

To obtain the path names for i-nodes 451 and 76 on a specified file system:

```
ff −i 451,76 /dev/rrp7
```

SEE ALSO

finc(8), find(1), frec(8), ncheck(8).

BUGS

Only a single path name out of any possible ones will be generated for a multiply linked i-node, unless the —l option is specified. When —l is specified, no selection criteria apply to the names generated. All possible names for every linked file on the file system will be included in the output.

On very large file systems, memory may run out before *ff* does.

NAME
filesave, tapesave − daily/weekly UNIX file system backup

SYNOPSIS
/etc/filesave.?
/etc/tapesave

DESCRIPTION
These shell scripts are provided as models. They are designed to provide a simple, interactive operator environment for file backup. *Filesave*.? is for daily disk-to-disk backup and *tapesave* is for weekly disk-to-tape.

The suffix .? can be used to name another system where two (or more) machines share disk drives (or tape drives) and one or the other of the systems is used to perform backup on both.

SEE ALSO
shutdown(8), volcopy(8).

NAME

format – how to format disks

SYNOPSIS

```
/bin/rxctrl -f
/sa[12]/rxformat
/sa[12]/rlformat
/sa[12]/rmformat
/sa[12]/emuformat
/sa[12]/xyloformat
```

DESCRIPTION

Rxctrl formats a 5 1/4" floppy emulating a single sided, double density 8" floppy with a ANDROMEDA WDC11 controller. For further details see *rxctrl*(1), the floppy driver manipulation program.

All the other formatting programs are standalone programs. Enter the Minitor (Type sync, push INIT) and type i.e.:

```
.rl         (load from RL02)
./sa1/emuformat (executable file)
.g0         (start the program)
```

Rxformat formats a 8" floppy RX02 compatible. The floppy controller responses '$'. Type

```
XD2 (double density) or
XD1 (single density) <cr>
   and
XU0 (left drive) or
XU1 (right drive) <cr>
```

Rlformat formats a whole TANDON TM603SE drive with a ANDROMEDA WDC11 controller as a RL02. The following arguments are interactively asked for:

```
UNIT:       0 (TANDON drive 0)
            1 (TANDON drive 1)
INTERLEAVE: 1..31 (odd)
            Use 5 to optimize the seek time.
```

Rmformat formats one or all tracks of a FUJITSU M2312K drive with a DATARAM S04/A controller as a RM02. The following arguments are interactively asked for:

```
SINGLE TRACK ?  'y' or <cr>
if yes:  HEAD:  0..4
         TRACK:  0..822
```

Emuformat formats a whole FUJITSU M2312K drive with a EMULEX SC02 controller as

```
Unit 0: RK07    Unit 3: RK07
Unit 1: RK07    Unit 4: RK07
Unit 2: RK06    Unit 5: RK06
```

The following argument is interactively asked for:

```
UNIT:  0..5
```

Xyloformat formats one or all tracks of a FUJITSU M2312K drive with a XYLOGICS 550 controller as

```
Unit 0: RK07
Unit 1: RK07
Unit 2: RK06
```

The following arguments are interactively asked for:

```
           UNIT:  0..2
SINGLE TRACK ?  'y' or <cr>
if yes   HEAD:  0..2
         TRACK:  0..410 for RK06
                 0..814 for RK07
```

SEE ALSO

rl(4), rx(4), hp(4), hk(4), mkfs(8)

BUGS

Old versions of the Minitor response the loading command with 'can't find file', if the loaded program is very small. Ignore this message!

NAME

fsba – file system block analyzer

SYNOPSIS

fsba file-system ...

DESCRIPTION

Fsba determines the number of extra sectors (1 sector has 512 bytes) needed when the file system logical block size is increased from 512 bytes per block to 1024 bytes/block. *File-system* should be specified by device name (e.g., **/dev/rhk2**).

Fsba determines how many sectors are currently allocated for the 512 bytes/block file system, and how many sectors will be required for the 1024 bytes/block converted file system. *Fsba* also prints out the number of allocated and free i-nodes for each *file system.*

If the number of free sectors for the 1024 bytes/block file system is negative, this indicates the file-system is too large to convert to 1024 bytes/block.

SEE ALSO

filesystem(5).

NAME

fsck, dfsck – file system consistency check and interactive repair

SYNOPSIS

/etc/fsck [**–y**] [**–n**] [**–sX**] [**–SX**] [**–t** file] [**–q**] [**–D**] [**–f**] [file-systems]

/etc/dfsck [options1] filsys1 ... **–** [options2] filsys2 ...

DESCRIPTION

Fsck

Fsck audits and interactively repairs inconsistent conditions for UNIX System files. If the file system is consistent then the number of files, number of blocks used, and number of blocks free are reported. If the file system is inconsistent the operator is prompted for concurrence before each correction is attempted. It should be noted that most corrective actions will result in some loss of data. The amount and severity of data lost may be determined from the diagnostic output. The default action for each consistency correction is to wait for the operator to respond **yes** or **no**. If the operator does not have write permission *fsck* will default to a **–n** action.

Fsck has more consistency checks than its predecessors *check*, *dcheck*, *fcheck*, and *icheck* combined.

The following options are interpreted by *fsck*.

–y Assume a yes response to all questions asked by *fsck*.

–n Assume a no response to all questions asked by *fsck*; do not open the file system for writing.

–s*X* Ignore the actual free list and (unconditionally) reconstruct a new one by rewriting the super-block of the file system. The file system should be unmounted while this is done; if this is not possible, care should be taken that the system is quiescent and that it is rebooted immediately afterwards. This precaution is necessary so that the old, bad, in-core copy of the superblock will not continue to be used, or written on the file system.

The **–s***X* option allows for creating an optimal free-list organization. The following forms of *X* are supported for the following devices:

–s3 (RP03)
–s4 (RP04, RP05, RP06)
–sBlocks-per-cylinder:Blocks-to-skip (for anything else)

If *X* is not given, the values used when the file system was created are used. If these values were not specified, then the value *400:7* is used.

–S*X* Conditionally reconstruct the free list. This option is like **–s***X* above except that the free list is rebuilt only if there were no discrepancies discovered in the file system. Using **–S** will force a no response to all questions asked by *fsck*. This option is useful for forcing free list reorganization on uncontaminated file systems.

–t If *fsck* cannot obtain enough memory to keep its tables, it uses a scratch file. If the **–t** option is specified, the file named in the next argument is used as the scratch file, if needed. Without the **–t** flag, *fsck* will prompt the operator for the name of the scratch file. The file chosen should not be on the file system being checked, and if it is not a special file or did not already exist, it is removed when *fsck* completes.

−q Quiet *fsck*. Do not print size-check messages in Phase 1. Unreferenced **fifos** will silently be removed. If *fsck* requires it, counts in the superblock will be automatically fixed and the free list salvaged.

−D Directories are checked for bad blocks. Useful after system crashes.

−f Fast check. Check block and sizes (Phase 1) and check the free list (Phase 5). The free list will be reconstructed (Phase 6) if it is necessary.

If no *file-systems* are specified, *fsck* will read a list of default file systems from the file **/etc/checklist**.

Inconsistencies checked are as follows:

1. Blocks claimed by more than one inode or the free list.
2. Blocks claimed by an inode or the free list outside the range of the file system.
3. Incorrect link counts.
4. Size checks:
   Incorrect number of blocks.
   Directory size not 16-byte aligned.
5. Bad inode format.
6. Blocks not accounted for anywhere.
7. Directory checks:
   File pointing to unallocated inode.
   Inode number out of range.
8. Super Block checks:
   More than 65536 inodes.
   More blocks for inodes than there are in the file system.
9. Bad free block list format.
10. Total free block and/or free inode count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the operator's concurrence, reconnected by placing them in the **lost+found** directory, if the files are nonempty. The user will be notified if the file or directory is empty or not. If it is empty, *fsck* will silently remove them. *Fsck* will force the reconnection of nonempty directories. The name assigned is the inode number. The only restriction is that the directory **lost+found** must preexist in the root of the file system being checked and must have empty slots in which entries can be made. This is accomplished by making **lost+found**, creating a number of files in the directory, and then removing them (before *fsck* is executed).

Checking the raw device is almost always faster and should be used with everything but the *root* file system.

Dfsck

Dfsck allows two file system checks on two different drives simultaneously. *options1* and *options2* are used to pass options to *fsck* for the two sets of file systems. A − is the separator between the file system groups.

The *dfsck* program permits an operator to interact with two *fsck*(8) programs at once. To aid in this, *dfsck* will print the file system name for each message to the operator. When answering a question from *dfsck*, the operator must prefix the response with a 1 or a 2 (indicating that the answer refers to the first or second file system group).

Do not use *dfsck* to check the *root* file system.

FILES

/etc/checklist	contains default list of file systems to check.
/etc/checkall	optimizing *dfsck* shell file.

SEE ALSO

checkall(8), clri(8), ncheck(8), checklist(5), fs(5).
Setting up the UNIX System

BUGS

Inode numbers for . and .. in each directory should be checked for validity.

DIAGNOSTICS

The diagnostics produced by *fsck* are intended to be self-explanatory.

NAME

fsdb – file system debugger

SYNOPSIS

/etc/fsdb special [–]

DESCRIPTION

Fsdb can be used to patch up a damaged file system after a crash. It has conversions to translate block and i-numbers into their corresponding disk addresses. Also included are mnemonic offsets to access different parts of an i-node. These greatly simplify the process of correcting control block entries or descending the file system tree.

Fsdb contains several error checking routines to verify i-node and block addresses. These can be disabled if necessary by invoking *fsdb* with the optional – argument or by the use of the **O** symbol. (*Fsdb* reads the i-size and f-size entries from the superblock of the file system as the basis for these checks.)

Numbers are considered decimal by default. Octal numbers must be prefixed with a zero. During any assignment operation, numbers are checked for a possible truncation error due to a size mismatch between source and destination.

Fsdb reads a block at a time and will therefore work with raw as well as block I/O. A buffer management routine is used to retain commonly used blocks of data in order to reduce the number of read system calls. All assignment operations result in an immediate write-through of the corresponding block.

The symbols recognized by *fsdb* are:

#	absolute address
i	convert from i-number to i-node address
b	convert to block address
d	directory slot offset
+,–	address arithmetic
q	quit
>,<	save, restore an address
=	numerical assignment
=+	incremental assignment
=–	decremental assignment
="	character string assignment
O	error checking flip flop
p	general print facilities
f	file print facility
B	byte mode
W	word mode
D	double word mode
!	escape to shell

The print facilities generate a formatted output in various styles. The current address is normalized to an appropriate boundary before printing begins. It advances with the printing and is left at the address of the last item printed. The output can be terminated at any time by typing the delete character. If a number follows the **p** symbol, that many entries are printed. A check is made to detect block boundary overflows since logically sequential blocks are generally not physically sequential. If a count of zero is used, all entries to the end of the current block are printed. The print options available are:

i	print as i-nodes
d	print as directories
o	print as octal words
e	print as decimal words
c	print as characters
b	print as octal bytes

The **f** symbol is used to print data blocks associated with the current i-node. If followed by a number, that block of the file is printed. (Blocks are numbered from zero.) The desired print option letter follows the block number, if present, or the **f** symbol. This print facility works for small as well as large files. It checks for special devices and that the block pointers used to find the data are not zero.

Dots, tabs and spaces may be used as function delimiters but are not necessary. A line with just a new-line character will increment the current address by the size of the data type last printed. That is, the address is set to the next byte, word, double word, directory entry or i-node, allowing the user to step through a region of a file system. Information is printed in a format appropriate to the data type. Bytes, words and double words are displayed with the octal address followed by the value in octal and decimal. A .B or .D is appended to the address for byte and double word values, respectively. Directories are printed as a directory slot offset followed by the decimal i-number and the character representation of the entry name. Inodes are printed with labeled fields describing each element.

The following mnemonics are used for i-node examination and refer to the current working i-node:

md	mode
ln	link count
uid	user ID number
gid	group ID number
sz	file size
a#	data block numbers (0 – 12)
at	access time
mt	modification time
maj	major device number
min	minor device number

EXAMPLES

386i	prints i-number 386 in an i-node format. This now becomes the current working i-node.
ln=4	changes the link count for the working i-node to 4.
ln=+1	increments the link count by 1.
fc	prints, in ASCII, block zero of the file associated with the working i-node.
2i.fd	prints the first 32 directory entries for the root i-node of this file system.

d5i.fc	changes the current i-node to that associated with the 5th directory entry (numbered from zero) found from the above command. The first logical block of the file is then printed in ASCII.
512.B.po	prints the superblock of this file system in octal.
2i.a0b.d7=3	changes the i-number for the seventh directory slot in the root directory to 3. This example also shows how several operations can be combined on one command line.
d7.nm="name"	changes the name field in the directory slot to the given string. Quotes are optional when used with **nm** if the first character is alphabetic.
a2b.p0d	prints the third block of the current inode as directory entries.

SEE ALSO

fsck(8), dir(5), fs(5).

NAME

fuser – identify processes using a file or file structure

SYNOPSIS

/etc/fuser [**–ku**] files [–] [[**–ku**] files]

DESCRIPTION

Fuser lists the process IDs of the processes using the *files* specified as arguments. For block special devices, all processes using any file on that device are listed. The process ID is followed by **c**, **p** or **r** if the process is using the file as its current directory, the parent of its current directory (only when in use by the system), or its root directory, respectively. If the **–u** option is specified, the login name, in parentheses, also follows the process ID. In addition, if the **–k** option is specified, the SIGKILL signal is sent to each process. Only the super-user can terminate another user's process (see *kill*(2)). Options may be respecified between groups of files. The new set of options replaces the old set, with a lone dash canceling any options currently in force.

The process IDs are printed as a single line on the standard output, separated by spaces and terminated with a single new line. All other output is written on standard error.

EXAMPLES

fuser –ku /dev/hk2
: will terminate all processes that are preventing disk drive hk2 from being unmounted if typed by the super-user, listing the process ID and login name of each as it is killed.

fuser –u /etc/passwd
: will list process IDs and login names of processes that have the password file open.

fuser –ku /dev/hk2 – –u /etc/passwd
: will do both of the above examples in a single command line.

FILES

/unix	for namelist
/dev/kmem	for system image
/dev/mem	also for system image

SEE ALSO

mount(8), ps(1), kill(2), signal(2).

NAME

fwtmp, wtmpfix – manipulate connect accounting records

SYNOPSIS

/usr/lib/acct/fwtmp [–ic]
/usr/lib/acct/wtmpfix [files]

DESCRIPTION

Fwtmp

Fwtmp reads from the standard input and writes to the standard output, converting binary records of the type found in **wtmp** to formatted ASCII records. The ASCII version is useful to enable editing, via *ed*(1), bad records or general purpose maintenance of the file.

The argument **–ic** is used to denote that input is in ASCII form, and output is to be written in binary form.

Wtmpfix

Wtmpfix examines the standard input or named files in **wtmp** format, corrects the time/date stamps to make the entries consistent, and writes to the standard output. A – can be used in place of *files* to indicate the standard input. If time/date corrections are not performed, *acctcon1* will fault when it encounters certain date change records.

Each time the date is set, a pair of date change records are written to **/etc/wtmp**. The first record is the old date denoted by the string **old time** placed in the line field and the flag **OLD_TIME** placed in the type field of the **<utmp.h>** structure. The second record specifies the new date and is denoted by the string **new time** placed in the line field and the flag **NEW_TIME** placed in the type field. *Wtmpfix* uses these records to synchronize all time stamps in the file.

In addition to correcting time/date stamps, *wtmpfix* will check the validity of the name field to ensure that it consists solely of alphanumeric characters, a **$** or spaces. If it encounters a name that is considered invalid, it will change the login name to **INVALID** and write a diagnostic to the standard error. In this way, *wtmpfix* reduces the chance that *acctcon1* will fail when processing connect accounting records.

FILES

/etc/wtmp
/usr/include/utmp.h

SEE ALSO

acct(8), acctcms(8), acctcom(1), acctcon(8), acctmerg(8), acctprc(8), acctsh(8), runacct(8), acct(2), acct(5), utmp(5).

NAME

getty – set terminal type, modes, speed, and line discipline

SYNOPSIS

/etc/getty [**–h**] [**–t** timeout] line [speed [type [linedisc]]]
/etc/getty –c file

DESCRIPTION

Getty is a program that is invoked by *init*(8). It is the second process in the series, (*init-getty-login-shell*) that ultimately connects a user with the UNIX System. Initially *getty* prints the login message field for the entry it is using from **/etc/gettydefs**. *Getty* reads the user's login name and invokes the *login*(1) command with the user's name as argument. While reading the name, *getty* attempts to adapt the system to the speed and type of terminal being used.

Line is the name of a tty line in **/dev** to which *getty* is to attach itself. *Getty* uses this string as the name of a file in the **/dev** directory to open for reading and writing. Unless *getty* is invoked with the **–h** flag, *getty* will force a hangup on the line by setting the speed to zero before setting the speed to the default or specified speed. The **–t** flag plus *timeout* in seconds, specifies that *getty* should exit if the open on the line succeeds and no one types anything in the specified number of seconds. The optional second argument, ***speed***, is a label to a speed and tty definition in the file **/etc/gettydefs**. This definition tells *getty* what speed to initially run at, what the login message should look like, what the inital tty settings are, and what speed to try next should the user indicate that the speed is inappropriate. (By typing a *<break>* character.) The default *speed* is 300 baud. The third argument, *type*, is a character string describing to *getty* what type of terminal is connected to the line in question.

The terminal type must be an entry in **/etc/termcap**. It is exported as "TERM=type" to a subsequent shell. The optional fourth argument, *linedisc*, is a character string describing which line discipline to use in communicating with the terminal. Again the hooks for line disciplines are available in the operating system but there is only one presently available, the default line discipline, **LDISC0**.

When given no optional arguments, *getty* sets the *speed* of the interface to 300 baud, specifies that raw mode is to be used (awaken on every character), that echo is to be suppressed, either parity allowed, newline characters will be converted to carriage return-line feed, and tab expansion performed on the standard output. It types the login message before reading the user's name a character at a time. If a null character (or framing error) is received, it is assumed to be the result of the user pushing the "break" key. This will cause *getty* to attempt the next *speed* in the series. The series that *getty* tries is determined by what it finds in **/etc/gettydefs**.

The user's name is terminated by a new-line or carriage-return character. The latter results in the system being set to treat carriage returns appropriately (see *ioctl*(2)).

The user's name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is non-empty, the system is told to map any future upper-case characters into the corresponding lower-case characters.

Finally, *login* is called with the user's name as an argument. Additional arguments may be typed after the login name. These are passed to *login*, which will place them in the environment (see *login*(1)).

A check option is provided. When *getty* is invoked with the **−c** option and *file*, it scans the file as if it were scanning **/etc/gettydefs** and prints out the results to the standard output. If there are any unrecognized modes or improperly constructed entries, it reports these. If the entries are correct, it prints out the values of the various flags. See *ioctl*(2) to interpret the values. Note that some values are added to the flags automatically.

FILES

/etc/gettydefs /etc/termcap

SEE ALSO

init(8), login(1), ioctl(2), gettydefs(5), inittab(5), termio(4)

NAME

init, telinit – process control initialization

SYNOPSIS

/etc/init [**0123456SsQq**]

/etc/telinit [**0123456sSQqabc**]

DESCRIPTION

Init

Init is a general process spawner. Its primary role is to create processes from a script stored in the file **/etc/inittab** (see *inittab*(5)). This file usually has *init* spawn *getty*'s on each line that a user may log in on. It also controls autonomous processes required by any particular system.

Init considers the system to be in a *run-level* at any given time. A *run-level* can be viewed as a software configuration of the system where each configuration allows only a selected group of processes to exist. The processes spawned by *init* for each of these *run-levels* is defined in the *inittab* file. *Init* can be in one of eight *run-levels*, **0–6** and S or **s**. The *run-level* is changed by having a privileged user run **/etc/init** (which is linked to */etc/telinit*). This user spawned *init* sends appropriate signals to the orginal *init* spawned by the operating system when the system was rebooted, telling it which *run-level* to change to.

Init is invoked inside the UNIX System as the last step in the boot procedure. The first thing *init* does is to look for **/etc/inittab** and see if there is an entry of the type *initdefault* (see *inittab*(5)). If there is, *init* uses the *run-level* specified in that entry as the initial *run-level* to enter. If this entry is not in *inittab* or *inittab* is not found, *init* requests that the user enter a *run-level* from the virtual system console, **/dev/syscon**. If an S (**s**) is entered, *init* goes into the *SINGLE USER* level. This is the only *run-level* that doesn't require the existence of a properly formatted *inittab* file. If **/etc/inittab** doesn't exist, then by default the only legal *run-level* that *init* can enter is the *SINGLE USER* level. In the *SINGLE USER* level the virtual console terminal **/dev/syscon** is opened for reading and writing and the command **/bin/su** is invoked immediately. To exit from the *SINGLE USER run-level* one of two options can be elected. First, if the shell is terminated (via an end-of-file), *init* will reprompt for a new *run-level*. Second, the *init* or *telinit* command can signal *init* and force it to change the *run-level* of the system.

When attempting to boot the system, failure of *init* to prompt for a new *run-level* may be due to the fact that the device **/dev/syscon** is linked to a device other than the physical system teletype (**/dev/systty**). If this occurs, *init* can be forced to relink **/dev/syscon** by typing a delete on the system teletype which is co-located with the processor.

When *init* prompts for the new *run-level*, the operator may only enter one of the digits **0** through **6** or the letters S or **s**. If S is entered *init* operates as previously described in *SINGLE USER* mode with the additional result that **/dev/syscon** is linked to the user's terminal line, thus making it the virtual system console. A message is generated on the physical console, **/dev/systty**, saying where the virtual terminal has been relocated.

When *init* comes up initially and whenever it switches out of *SINGLE USER* state to normal run states, it sets the *ioctl*(2) states of the virtual console, **/dev/syscon**, to those modes saved in the file **/etc/ioctl.syscon**. This file is written by *init* whenever *SINGLE USER* mode is entered. If this file doesn't exist when *init* wants to read it, a warning is printed and default settings are assumed.

If a **0** through **6** is entered *init* enters the corresponding *run-level*. Any other input will be rejected and the user will be re-prompted. If this is the first time *init* has entered a *run-level* other than *SINGLE USER*, *init* first scans *inittab* for special entries of the type *boot* and *bootwait*. These entries are performed, providing the *run-level* entered matches that of the entry before any normal processing of *inittab* takes place. In this way any special initialization of the operating system,such as mounting file systems, can take place before users are allowed onto the system. The *inittab* file is scanned to find all entries that are to be processed for that *run-level*.

Run-level 2 is usually defined by the user to contain all of the terminal processes and daemons that are spawned in the multi-user environment.

In a multi-user environment, the *inittab* file is usually set up so that *init* will create a process for each terminal on the system.

For terminal processes, ultimately the shell will terminate because of an end-of-file either typed explicitly or generated as the result of hanging up. When *init* receives a child death signal, telling it that a process it spawned has died, it records the fact and the reason it died in **/etc/utmp** and **/etc/wtmp** if it exists (see *who*(1)). A history of the processes spawned is kept in **/etc/wtmp** if such a file exists.

To spawn each process in the *inittab* file, *init* reads each entry and for each entry which should be respawned, it forks a child process. After it has spawned all of the processes specified by the *inittab* file, *init* waits for one of its descendant processes to die, a powerfail signal, or until *init* is signaled by *init* or *telinit* to change the system's *run-level*. When one of the above three conditions occurs, *init* re-examines the *inittab* file. New entries can be added to the *inittab* file at any time; however, *init* still waits for one of the above three conditions to occur. To provide for an instantaneous response the **init Q** or **init q** command can wake *init* to re-examine the *inittab* file.

If *init* receives a *powerfail* signal (*SIGPWR*) and is not in *SINGLE USER* mode, it scans *inittab* for special powerfail entries. These entries are invoked (if the *run-levels* permit) before any further processing takes place. In this way *init* can perform various cleanup and recording functions whenever the operating system experiences a power failure. It is important to note that the powerfail entries should not use devices that must first be initialized (e.g. dzb lines) after a power failure has occurred.

When *init* is requested to change *run-levels* (via *telinit*), *init* sends the warning signal (**SIGTERM**) to all processes that are undefined in the target *run-level*. *Init* waits 20 seconds before forcibly terminating these processes via the kill signal (**SIGKILL**).

Telinit

Telinit, which is linked to */etc/init*, is used to direct the actions of *init*. It takes a one character argument and signals *init* via the kill system call to perform the appropriate action. The following arguments serve as directives to *init*.

0–6	tells *init* to place the system in one of the *run-levels* **0–6**.
a,b,c	tells *init* to process only those **/etc/inittab** file entries having the **a**, **b** or **c** *run-level* set.
Q,q	tells *init* to re-examine the **/etc/inittab** file.
s,S	tells *init* to enter the single user environment. When this level change is effected, the virtual system teletype, **/dev/syscon**, is changed to the terminal from which the command was executed.

Telinit can only be run by someone who is super-user or a member of group **sys**.

FILES

/etc/inittab
/etc/utmp
/etc/wtmp
/etc/ioctl.syscon
/dev/syscon
/dev/systty

SEE ALSO

getty(8), login(1), sh(1), who(1), kill(2), inittab(5), utmp(5).

DIAGNOSTICS

If *init* finds that it is continuously respawning an entry from **/etc/inittab** more than 10 times in 2 minutes, it will assume that there is an error in the command string, and generate an error message on the system console, and refuse to respawn this entry until either 5 minutes has elapsed or it receives a signal from a user *init* (*telinit*). This prevents *init* from eating up system resources when someone makes a typographical error in the *inittab* file or a program is removed that is referenced in the *inittab*.

NAME

install – install commands

SYNOPSIS

/etc/install [–c dira] [–f dirb] [–i] [–n dirc] [–o] [–s] file [dirx ...]

DESCRIPTION

Install is a command most commonly used in "makefiles" (see *make*(1)) to install a *file* (updated target file) in a specific place within a file system. Each *file* is installed by copying it into the appropriate directory, thereby retaining the mode and owner of the original command. The program prints messages telling the user exactly what files it is replacing or creating and where they are going.

If no options or directories (*dirx* ...) are given, *install* will search a set of default directories (**/bin**, **/usr/bin**, **/etc**, **/lib**, and **/usr/lib**, in that order) for a file with the same name as *file*. When the first occurrence is found, *install* issues a message saying that it is overwriting that file with *file*, and proceeds to do so. If the file is not found, the program states this and exits without further action.

If one or more directories (*dirx* ...) are specified after *file*, those directories will be searched before the directories specified in the default list.

The meanings of the options are:

–c *dira* Installs a new command (*file*) in the directory specified by *dira*, only if it is not found. If it is found, *install* issues a message saying that the file already exists, and exits without overwriting it. May be used alone or with the **–s** option.

–f *dirb* Forces *file* to be installed in given directory, whether or not one already exists. If the file being installed does not already exist, the mode and owner of the new file will be set to 755 and **bin**, respectively. If the file already exists, the mode and owner will be that of the already existing file. May be used alone or with the **–o** or **–s** options.

–i Ignores default directory list, searching only through the given directories (*dirx* ...). May be used alone or with any other options other than **–c** and **–f**.

–n *dirc* If *file* is not found in any of the searched directories, it is put in the directory specified in *dirc*. The mode and owner of the new file will be set to 755 and **bin**, respectively. May be used alone or with any other options other than **–c** and **–f**.

–o If *file* is found, this option saves the "found" file by copying it to **OLD***file* in the directory in which it was found. This option is useful when installing a normally text busy file such as */bin/sh* or */etc/getty*, where the existing file cannot be removed. May be used alone or with any other options other than **–c**.

−s Suppresses printing of messages other than error messages. May be used alone or with any other options.

SEE ALSO
make(1).

NAME
killall – kill all active processes

SYNOPSIS
/etc/killall [signal]

DESCRIPTION
Killall is is a procedure used by **/etc/shutdown** to kill all active processes not directly related to the shut down procedure.

Killall is chiefly used to terminate all processes with open files so that the mounted file systems will be unbusied and can be unmounted.

Killall sends *signal* (see *kill*(1)) to all remaining processes not belonging to the above group of exclusions. If no *signal* is specified, a default of 9 is used.

FILES
/etc/shutdown

SEE ALSO
fuser(8), kill(1), ps(1), shutdown(8), signal(2).

NAME
: link, unlink – exercise link and unlink system calls

SYNOPSIS
: **/etc/link** file1 file2
: **/etc/unlink** file

DESCRIPTION
: *Link* and *unlink* perform their respective system calls on their arguments, abandoning all error checking. These commands may only be executed by the super-user, who (it is hoped) knows what he or she is doing.

SEE ALSO
: rm(1), link(2), unlink(2).

NAME

lpd – line printer daemon

SYNOPSIS

/usr/lib/lpd

DESCRIPTION

Lpd is the daemon for the line printer. *Lpd* uses the directory */usr/spool/lpd*. The file *lock* in that directory is used to prevent two daemons from becoming active. After the program has successfully set the lock, it forks and the main path exits, thus spawning the daemon. The directory is scanned for files beginning with **df**. Each such file is submitted as a job. Each line of a job file must begin with a key character to specify what to do with the remainder of the line.

L	specifies that the remainder of the line is to be sent as a literal.
B	specifies that the rest of the line is a file name.
F	is the same as **B** except a form feed is prepended to the file.
U	specifies that the rest of the line is a file name. After the job has been transmitted, the file is unlinked.
M	is followed by a user ID; after the job is sent, a message is mailed to the user via the *mail*(1) command to verify the sending of the job.

Any error encountered will cause the daemon to wait and start over. This means that an improperly constructed **df** file may cause the same job to be submitted repeatedly.

Lpd is automatically initiated by the line printer command, *lpr*.

To restart *lpd* (in the case of hardware or software malfunction), it is necessary to first kill the old daemon (if still alive), and remove the lock file before initiating the new daemon. This is done automatically when the system is brought up, by */etc/rc*, in case there were any jobs left in the spooling directory when the system last went down.

FILES

/usr/spool/lpd/*	spool area for line printer daemon
/etc/passwd	to get the user's name
/dev/lp	line printer device

SEE ALSO

lpr(1)

NAME

makekey – generate encryption key

SYNOPSIS

/usr/lib/makekey

DESCRIPTION

Makekey improves the usefulness of encryption schemes depending on a key by increasing the amount of time required to search the key space. It reads 10 bytes from its standard input, and writes 13 bytes on its standard output. The output depends on the input in a way intended to be difficult to compute (i.e. to require a substantial fraction of a second).

The first eight input bytes (the *input key*) can be arbitrary ASCII characters. The last two (the *salt*) are best chosen from the set of digits, upper- and lower-case letters, and '.' and '/'. The salt characters are repeated as the first two characters of the output. The remaining 11 output characters are chosen from the same set as the salt and constitute the *output key*.

The transformation performed is essentially the following: the salt is used to select one of 4096 cryptographic machines all based on the National Bureau of Standards DES algorithm, but modified in 4096 different ways. Using the input key as key, a constant string is fed into the machine and recirculated a number of times. The 64 bits that come out are distributed into the 66 useful key bits in the result.

Makekey is intended for programs that perform encryption (e.g. *ed* and *crypt*(1)). Usually its input and output will be pipes.

SEE ALSO

crypt(1), ed(1)

NAME

/etc/mkalias – create an alias to a remote file

SYNOPSIS

/etc/mkalias [-f] name identifier

DESCRIPTION

This program facilitates the creation of "aliases" to remote files. An alias is a name that appears to be local but in fact refers to a file of the same name on another system. Thus a system with no line printer could have a file "/dev/lp" which was an alias for the line printer on another system, (where it would have to be called "/dev/lp" as well). The operation of an alias is therefore to take the name provided by the user and attempt to perform the requested operation on the file of that name on the indicated system. The "identifier" parameter is of the same type as that provided to *mksys*(8N): for Release 1.0, it must be an integer in the range [0..255] inclusive. It will be passed to your network interface to identify the name neighbour system on which the "real" version of this file appears.

BUGS

Aliased files must always be accessed relative to "/", otherwise they will not be found.

DIAGNOSTICS

Complains if file "name" exists, unless the "-f" option is present.

NAME

mkfs – construct a file system

SYNOPSIS

/etc/mkfs special blocks[:inodes] [gap blocks/cyl]
/etc/mkfs special proto [gap blocks/cyl]

DESCRIPTION

Mkfs constructs a file system by writing on the special file according to the directions found in the remainder of the command line. If the second argument is given as a string of digits, *mkfs* builds a file system with a single empty directory on it. The size of the file system is the value of *blocks* interpreted as a decimal number. This is the number of *physical* disk blocks the file system will occupy. The boot program is left uninitialized. If the optional number of inodes is not given, the default is the number of *logical* blocks divided by 4.

If the second argument is a file name that can be opened, *mkfs* assumes it to be a prototype file *proto*, and will take its directions from that file. The prototype file contains tokens separated by spaces or new-lines. The first token is the name of a file to be copied onto block zero as the bootstrap program (see *boot*(8)). The second token is a number specifying the size of the created file system in *physical* disk blocks. Typically it will be the number of blocks on the device, perhaps diminished by space for swapping. The next token is the number of inodes in the file system. The maximum number of inodes configurable is 65500. The next set of tokens comprise the specification for the root file. File specifications consist of tokens giving the mode, the user ID, the group ID, and the initial contents of the file. The syntax of the contents field depends on the mode.

The mode token for a file is a 6 character string. The first character specifies the type of the file. (The characters **–bcd** specify regular, block special, character special and directory files respectively.) The second character of the type is either **u** or **–** to specify set-user-id mode or not. The third is **g** or **–** for the set-group-id mode. The rest of the mode is a three digit octal number giving the owner, group, and other read, write, execute permissions (see *chmod*(1)).

Two decimal number tokens come after the mode; they specify the user and group ID's of the owner of the file.

If the file is a regular file, the next token is a path name whence the contents and size are copied. If the file is a block or character special file, two decimal number tokens follow which give the major and minor device numbers. If the file is a directory, *mkfs* makes the entries . and .. and then reads a list of names and (recursively) file specifications for the entries in the directory. The scan is terminated with the token **$**.

A sample prototype specification follows:

```
/stand/diskboot
4872 110
d––777 3 1
usr     d––777 3 1
        sh      –––755 3 1 /bin/sh
        ken     d––755 6 1
```

```
                    $
            b0      b--644 3 1 0 0
            c0      c--644 3 1 0 0
            $
        $
```

In both command syntaxes, the rotational *gap* and the number of *blocks/cyl* can be specified. The following values are recommended:

Device	Gap Size	Blks/Cyl
RL01/02	7	40
RP03	5	200
RP04/05/06	7	418
RP07	7	400
RM03	7	160
RM05	7	608
RM80	9	434
default	7	400

The *default* will be used if the supplied *gap* and *blocks/cyl* are considered illegal values or if a short argument count occurs.

FILES

/etc/mkfs	for 1024 byte block filesystem
/etc/mkfs.1b	for 512 byte block filesystem

SEE ALSO

dir(5), fs(5), boot(8).

BUGS

If a prototype is used, it is not possible to initialize a file larger than 64K bytes, nor is there a way to specify links.

NAME

mknod – build special file

SYNOPSIS

/etc/mknod name **c** | **b** major minor
/etc/mknod name **p**

DESCRIPTION

Mknod makes a directory entry and corresponding i-node for a special file. The first argument is the *name* of the entry. In the first case, the second is **b** if the special file is block-type (disks, tape) or **c** if it is character-type (other devices). The last two arguments are numbers specifying the *major* device type and the *minor* device (e.g. unit, drive, or line number), which may be either decimal or octal.

The assignment of major device numbers is specific to each system. They have to be dug out of the system source file **/usr/sys/c.c**. The letter **b** refers to the array bdevsw, the letter **c to the array cdevsw in** The major number is the row number of the corresonding entry.

Mknod can also be used to create fifo's (a.k.a named pipes) (second case in *SYNOPSIS* above).

SEE ALSO

mknod(2).

NAME

/etc/mksys – make a remote system node

SYNOPSIS

/etc/mksys [-p] | [[-f] name identifier ethernet-address]

DESCRIPTION

This program is used to create the special directory entries needed to communicate with remote systems via the Newcastle Connection. The first parameter is the name that the new entry is to have, and the second is the identifier of the system it is to refer to. The identifier must be in the range [0..255] inclusive for Release 1.0 of the Newcastle Connection. It will be passed to your network interface via the procedure "_netitoa()" to be converted into a physical address when required. The inverse function "_netatoi()" will be called by the Connection to translate a network address into an identifier. Depending on your network interface, it may be possible to encode "identifier" so that it is particularly easy to transform it to a physical address. For example, "identifier" can be used directly as a station address for a Cambridge Ring. The ethernet address is a 6 byte hexadecimal number, given in 12 characters in the range [0-9a-f]. The station identifier and the ethernet address of a remote machine must be consistent with this machines' declarations in /usr/sys/name.c

The "-p" option causes "mksys" to print the list of name-identifier pairs known to the local system from the file "/etc/utab", plus the corresponding ethernet addresses from the file "/etc/map_port_eadr".

"Mksys" normally complains if "name" already exists. This can be overridden by the "-f" option.

FILES

/etc/utab - table of name-identifier pairs
/etc/map_port_eadr - table of ethernet addresses, indexed by identifier

EXAMPLE

with virtual superroot:

```
        /etc/mksys /../alpha 4 12440a1b041e
```

without virtual superroot:

```
        /etc/mksys /alpha 4 12440a1b041e
```

SEE ALSO

"The Newcastle Connection – Release 1.0: Network Interface Installation Guide", rmsys(8N), utab(5N)

NAME

mount, umount – mount and dismount file system

SYNOPSIS

/etc/mount [special directory [**−r**]]

/etc/umount special

DESCRIPTION

Mount announces to the system that a removable file system is present on the device *special*. The *directory* must exist already; it becomes the name of the root of the newly mounted file system.

These commands maintain a table of mounted devices. If invoked with no arguments, *mount* prints the table.

The optional last argument indicates that the file is to be mounted read-only. Physically write-protected and magnetic tape file systems must be mounted in this way or errors will occur when access times are updated, whether or not any explicit write is attempted.

Umount announces to the system that the removable file system previously mounted on device *special* is to be removed.

FILES

/etc/mnttab mount table

SEE ALSO

setmnt(8), mount(2), mnttab(5).

DIAGNOSTICS

Mount issues a warning if the file system to be mounted is currently mounted under another name.

Umount complains if the special file is not mounted or if it is busy. The file system is busy if it contains an open file or some user's working directory.

BUGS

Some degree of validation is done on the file system, however it is generally unwise to mount garbage file systems.

NAME

mvdir – move a directory

SYNOPSIS

/etc/mvdir dirname name

DESCRIPTION

Mvdir renames directories within a file system. *Dirname* must be a directory; *name* must not exist. Neither name may be a sub-set of the other (**/x/y** cannot be moved to **/x/y/z**, nor vice versa).

Only super-user can use *mvdir*.

SEE ALSO

mkdir(1).

NAME

ncheck – generate names from i-numbers

SYNOPSIS

/etc/ncheck [**–i** numbers] [**–a**] [**–s**] [file-system]

DESCRIPTION

Ncheck with no argument generates a path name vs. i-number list of all files on a set of default file systems. Names of directory files are followed by /.. The **–i** option reduces the report to only those files whose i-numbers follow. The **–a** option allows printing of the names . and .., which are ordinarily suppressed. The **–s** option reduces the report to special files and files with set-user-ID mode; it is intended to discover concealed violations of security policy.

A file system may be specified.

The report is in no useful order, and probably should be sorted.

SEE ALSO

fsck(8), sort(1).

DIAGNOSTICS

When the file system structure is improper, ?? denotes the "parent" of a parentless file and a path name beginning with ... denotes a loop.

NAME

/etc/NCsetup − initialise the Newcastle Connection tables in a process

SYNOPSIS

/usr/NCbin/NCsetup

DESCRIPTION

This program initialises the "__N" environment string used by the Newcastle Connection before executing the shell. It should be named as a user's shell in the file "/etc/passwd", for all those users who are to have immediate access to the Connection. Otherwise, **/bin/NCon** may be called, which is a small shell script that calls **/usr/NCbin/NCsetup**. The Newcastle Connection can only be used when the "__N" string is in the environment.

NAME

newconf – generate configuration file and reconfigure **MUNIX**

SYNOPSIS

/etc/newconf

DESCRIPTION

Newconf creates the files conf.h, conf.modul and name.c needed to make a new **MUNIX** kernel.

Newconf asks for

– the device drivers to be included
(some drivers are automatically included)

– whether controllers are 18 or 22 bit DMA controllers

– assignation of DMA extension registers to DMA devices

– the type and unit of the root and swap device

– the origin (block number) and size of the swap area

– some other system parameters

Newconf creates the configuration file /usr/sys/conf.h as include file to the configuration table /usr/sys/c.c and the interrupt vector table /usr/sys/l.s.

The file /usr/sys/conf.modul contains the names of the driver modules, which are to be extracted from /usr/sys/libchoice to form the library /usr/sys/lib3. The library libchoice contains pairs of drivers for 18 and 22 bit DMA devices. E.g. a controller for the RK07 may come as an (old) 18 bit or a (new) 22 bit wide device. Correspondingly there exist drivers hk18.o and hk22.o in libchoice, one of which must be chosen.

The file /usr/sys/name.c contains the identification of your system. Most important are the nodename and the ethernet address. Newconf will ask for them, other fields you have to edit yourself.

It is instructive to read the files /etc/newconf, /usr/sys/c.c, /usr/sys/l.s, /usr/sys/name.c and /usr/sys/makefile. You should also look at the header files (in /usr/include/sys) param.h, types.h, sysmacros.h and space.h.

Reconfiguration of **MUNIX** is done by invoking "make" in directory /usr/sys. This will result in compiling /usr/sys/c.c, /usr/sys/name.c, assembling /usr/sys/l.s and linking /usr/sys/c.o, /usr/sys/l.o, /usr/sys/name.o, and others, the kernel library /usr/sys/lib1 and the driver libraries /usr/sys/lib2 and /usr/sys/lib3 to a new MUNIX kernel named */nunix*.

FILES

/usr/sys/conf.h
/usr/sys/c.*
/usr/sys/name.*
/usr/sys/l.*
/usr/sys/lib*
/nunix

SEE ALSO

whatconf (8)

NAME

mknf, rmnf, nfxmit, nfrcv, nfarchive, newsinput, newsoutput – notesfile utility programs

SYNOPSIS

mknf [−aon] topic [...]

rmnf topic [...]

nfxmit -dsite [-r] [-a] [-f file] topic [...]

nfrcv topic fromsystem

nfarchive [-n] [-d] [-f file] topic [...]

newsinput

newsoutput [-a] [-f file] topic [...]

DESCRIPTION

mknf, *rmnf*, *nfxmit*, *nfrcv*, *nfarchive*, *newsinput*, and *newwsoutput* are the utility programs provided with the notesfile system. They provide the capabilities to create and remove notesfiles, update intersystem notesfiles, archive old notes, and perform gateway activity between news(I) and the notesfile system.

mknf and *rmnf* create and delete notesfiles respectively. The same parameters apply for each: the 'topic' is the name that the notesfile is known by. As *mknf* processes its arguments, creating new notesfiles, the name of each new notesfile is echoed to the terminal. The new notesfiles are closed and the *notesfile owner* is made the sole director. He customarily turns control over to the user requesting the notesfile by making that person a director. The *-aon* options apply to *mknf* only. They signify that the notesfiles created are to permit anonymous notes, be open, and be networked respectively.

rmnf asks for verification of each notesfile before deleting it. The notesfile is deleted if the response line begins with a 'y'. Only the *notesfile owner* is allowed to run *mknf* and *rmnf*.

Network transmission of notesfiles is accomplished using *nfxmit* and *nfrcv*. *Nfxmit* sends the specified notesfiles to *site*. The *-r* option specifies that a request should be queued for the remote site to transmit updates from its copies of the notesfiles sent. Specify *-a* to have articles which originated in news(1) sent. News(1)-originated articles will usually reach each system via the news program. A timestamp of the last transmission of each notesfile to each system is maintained. This is used for determining the notes to send. The *-r* option is used only if the other site does not automatically queue updates of the notesfile.

Specify *-f myfile* on the command line to have *nfxmit* read *myfile* for a list of notesfiles to be sent. This is useful if the number of notesfiles is too numerous to list on a single command line. The shell metacharacters *, ?, [, and] are recognized in both the *topic* parameter and the entries in *myfile*.

Nfxmit uses uux(I) to invoke *nfrcv* on the remote system in order to process the incoming notes. Non-uucp connections are also supported.

Nfarchive is used to archive notes that have not been modified in a certain amount of time. The *-n* parameter gives the number of days that a note must be unmodified before being eligible for archival. The archived notes are stored in a 'generic' format in a separate directory where they are available for later retrieval. The *-f* parameter is similar to that of the *nfxmit* parameter of the same name. The *-d* parameter tells *nfarchive* that the eligible notes should be deleted only; they are not placed into the archives.

To transfer from news(1) to notesfiles, arrange to have the news distribution program forward articles it receives to *newsinput*. *Newsinput* parses the A news protocol for intersystem transfer. See the *Notesfile Reference Manual* for more detail on how to establish this connection.

Newsoutput takes several arguments. The *-a* option specifies that notesfile originated articles from other systems can be sent to news; the default is that only locally written articles are passed to news. Use the *-f* option to specify a file which contains a list of notesfiles to process. The rest of the command line contains notesfiles whose new articles are spooled to news.

The file '/usr/spool/notes/.utilities/newsgroups', if present, contains mapping functions between notesfiles and newsgroups. The mapping permits several newsgroups to be tied to a notesfile. The file format is:

```
notesfile:newsgroup
```

More information on this feature can be found in *The Notesfile Reference Manual*.

BUGS

The arciver does not have a matching unarchiver. To recover unarchived notes, one has to feed the archive into the nfrcv program.

If several systems sharing a common notesfile all decide to run newsoutput with the -a option for that notesfile, duplicate articles may appear in the news(1) system.

Newsinput is naive about parsing author names. In particular it gets confused with mixtures of UUCP and ARPA addressing.

The news/notes software puts lines into news articles which start "#:". Some mail programs dislike this, even though it comes after the blank line which separates the header from the text. Once large numbers of the sites running news have the release which passes unrecognized header lines, a change will be made to move the line into the headers. This should make many people happier.

FILES

/usr/spool/notes/.utilities	where most of these programs live.
/usr/spool/notes/.utilities/newsgroups	mapping betwcen notesfiles and newsgroups
/usr/spool/notes/.utilities/net.how	specifies connection methods between systems
/usr/spool/notes/.utilities/net.alias	directory containing mapping of local and remote notesfile names

SEE ALSO
news(1), notes(1), nfcomment(3), uucp(1C),
The Notesfile Reference Manual

AUTHORS
Ray Essick (uiucdcs!essick, uiucdcs!notes)
Rob Kolstad (uiucdcs!kolstad)
Department of Computer Science
222 Digital Computer Laboratory
University of Illinois at Urbana-Champaign
1304 West Springfield Ave.
Urbana, IL 61801

NAME

pstat – print system facts

SYNOPSIS

pstat [**–aixptuf**] [suboptions] [file]

DESCRIPTION

Pstat interprets the contents of certain system tables. If *file* is given, the tables are sought there, otherwise in */dev/mem*. The required namelist is taken from */unix*. Options are

–a Under **–p**, describe all process slots rather than just active ones.

–i Print the inode table with the these headings:

LOC	The core location of this table entry.
FLAGS	Miscellaneous state variables encoded thus:

L	locked
U	update time *filesystem*(5)) must be corrected
A	access time must be corrected
M	file system is mounted here
W	wanted by another process (L flag is on)
T	contains a text file
C	changed time must be corrected

CNT	Number of open file table entries for this inode.
DEVICE	Major and minor device number of file system in which this inode resides.
INO	I-number within the device.
MODE	Mode bits, see *chmod*(2).
NLK	Number of links to this inode.
UID	User ID of owner.
SIZ/DEV	Number of bytes in an ordinary file, or major and minor device of special file.

–x Print the text table with these headings:

LOC	The core location of this table entry.
FLAGS	Miscellaneous state variables encoded thus:

T	*ptrace*(2) in effect
W	text not yet written on swap device
L	loading in progress
K	locked
W	wanted (L flag is on)

DADDR	Disk address in swap, measured in multiples of 512 bytes.
CADDR	Core address, measured in multiples of 512 bytes.
SIZE	Size of text segment, measured in multiples of 512 bytes.
IPTR	Core location of corresponding inode.
CNT	Number of processes using this text segment.
CCNT	Number of processes in core using this text segment.

–p Print process table for active processes with these headings:

LOC	The core location of this table entry.
S	Run state encoded thus:

	0	no process
	1	waiting for some event
	3	runnable
	4	being created
	5	being terminated
	6	stopped under trace

F	Miscellaneous state variables, or-ed together:

	01	loaded
	02	the scheduler process
	04	locked
	010	swapped out
	020	traced
	040	used in tracing
	0100	locked in by *plock*(2).

PRI	Scheduling priority, see *nice*(2).
SIGNAL	Signals received (signals 1-16 coded in bits 0-15),
UID	Real user ID.
TIM	Time resident in seconds; times over 127 coded as 127.
CPU	Weighted integral of CPU time, for scheduler.
NI	Nice level, see *nice*(2).
PGRP	Process number of root of process group (the opener of the controlling terminal).
PID	The process ID number.
PPID	The process ID of parent process.
ADDR	If in core, the physical address of the 'u-area' of the process, if swapped out, the position in the swap area, each time measured in multiples of 512 bytes.
SIZE	Size of process image in multiples of 512 bytes.
WCHAN	Wait channel number of a waiting process.
LINK	Link pointer in list of runnable processes.
TEXTP	If text is pure, pointer to location of text table entry.
CLKT	Countdown for *alarm*(2) measured in seconds.

−t Print table for terminals with these headings:

RAW	Number of characters in raw input queue.
CAN	Number of characters in canonicalized input queue.
OUT	Number of characters in output queue.
IFLAG	See *termio*(4). The same for OFLAG, CFLAG, LFLAG.
STATE	See */usr/include/sys/tty.h*(internal*state*flags).
DEL	Number of delimiters (newlines) in canonicalized input queue.
COL	Calculated column position of terminal.
PGRP	Process group for which this is controlling terminal.

−u print information about a user process; the next argument is its address as given by *ps*(1). The process must be in main memory, or the file used can be a core image and the address 0.

−f Print the open file table with these headings:

LOC	The core location of this table entry.

FLG	Miscellaneous state variables encoded thus:	
	R	open for reading
	W	open for writing
	P	pipe
CNT	Number of processes that know this open file.	
INO	The location of the inode table entry for this file.	
OFFS	The file offset, see *lseek*(2).	

FILES

/unix	namelist
/dev/mem	default source of tables

SEE ALSO

ps(1), stat(2), fs(5)
K. Thompson, *UNIX Implementation*

NAME

pwck, grpck – password/group file checkers

SYNOPSIS

/etc/pwck [file]
/etc/grpck [file]

DESCRIPTION

Pwck scans the password file and notes any inconsistencies. The checks include validation of the number of fields, login name, user ID, group ID, and whether the login directory and optional program name exist. The default password file is **/etc/passwd**.

Grpck verifies all entries in the group file. This verification includes a check of the number of fields, group name, group ID, and whether all login names appear in the password file. The default group file is **/etc/group**.

FILES

/etc/group
/etc/passwd

SEE ALSO

group(5), passwd(5).
Setting up the UNIX System

DIAGNOSTICS

Group entries in **/etc/group** with no login names are flagged.

NAME

quot – summarize file system ownership

SYNOPSIS

quot [option] ... [filesystem]

DESCRIPTION

Quot prints the number of blocks in the named *filesystem* currently owned by each user. If no *filesystem* is named, all mounted filesystems will be scanned. The following options are available:

–n Cause the pipeline **ncheck filesystem | sort +0n | quot –n filesystem** to produce a list of all files and their owners.

–c Print three columns giving file size in blocks, number of files of that size, and cumulative total of blocks in that size or smaller file.

–f Print count of number of files as well as space owned by each user. /etc/passwd to get user names

SEE ALSO

ls(1), du(1)

BUGS

Holes in files are counted as if they actually occupied space.

NAME

recnews − receive unprocessed articles via mail

SYNOPSIS

/usr/lib/news/recnews [*newsgroup* [*sender*]]

DESCRIPTION

Recnews reads a letter from the standard input; determines the article title, sender, and newsgroup; and gives the body to inews with the right arguments for insertion.

If *newsgroup* is omitted, the to line of the letter will be used. If *sender* is omitted, the sender will be determined from the from line of the letter. The title is determined from the subject line.

SEE ALSO

inews(1), uurec(8), sendnews(8), readnews(1), checknews(1)

NAME

renice – alter priority of running process by changing nice

SYNOPSIS

/etc/renice pid [priority]

DESCRIPTION

Renice can be used by the super-user to alter the priority of a running process. By default, the nice of the process is made 19 which means that it will run only when nothing else in the system wants to. This can be used to nail long running processes which are interfering with interactive work.

Renice can be given a second argument to choose a nice other than the default. Negative nices can be used to make things go very fast.

FILES

/unix
/dev/kmem

SEE ALSO

nice(1)

BUGS

If you make the nice very negative, then the process cannot be interrupted. To regain control you must put the nice back (e.g. to 0.)

NAME

restor – incremental file system restore

SYNOPSIS

restor key [argument ...]

DESCRIPTION

Restor is used to read magtapes dumped with the *dump* command. The *key* specifies what is to be done. *Key* is one of the characters **rRxt** optionally combined with **f**.

f Use the first *argument* as the name of the tape instead of the default.

r or R The tape is read and loaded into the file system specified in *argument*. This should not be done lightly (see below). If the key is R *restor* asks which tape of a multi volume set to start on. This allows restor to be interrupted and then restarted (an *icheck* –sor*fsck* must be done before restart).

x Each file on the tape named by an *argument* is extracted. The file name has all 'mount' prefixes removed; for example, /usr/bin/lpr is named /bin/lpr on the tape. The file extracted is placed in a file with a numeric name supplied by *restor* (actually the inode number). In order to keep the amount of tape read to a minimum, the following procedure is recommended:

Mount volume 1 of the set of dump tapes.

Type the *restor* command.

Restor will announce whether or not it found the files, give the number it will name the file, and rewind the tape.

It then asks you to 'mount the desired tape volume'. Type the number of the volume you choose. On a multivolume dump the recommended procedure is to mount the last through the first volume in that order. *Restor* checks to see if any of the files requested are on the mounted tape (or a later tape, thus the reverse order) and doesn't read through the tape if no files are. If you are working with a single volume dump or the number of files being restored is large, respond to the query with '1' and *restor* will read the tapes in sequential order.

If you have a hierarchy to restore you can use dumpdir(8) to produce the list of names and a shell script to move the resulting files to their homes.

t Print the date the tape was written and the date the filesystem was dumped from.

The **r** option should only be used to restore a complete dump tape onto a clear file system or to restore an incremental dump tape onto this. Thus

```
/etc/mkfs /dev/rp0 40600
restor r /dev/rp0
```

is a typical sequence to restore a complete dump. Another *restor* can be done to get an incremental dump in on top of this.

A *dump* followed by a *mkfs* and a *restor* is used to change the size of a file system.

FILES

default tape unit varies with installation
restor.1b restor for 512byte filesystems
rst*

SEE ALSO

dump(8), mkfs(8), dumpdir(8)

DIAGNOSTICS

There are various diagnostics involved with reading the tape and writing the disk. There are also diagnostics if the i-list or the free list of the file system is not large enough to hold the dump.

If the dump extends over more than one tape, it may ask you to change tapes. Reply with a new-line when the next tape has been mounted.

RE1024

For conversion purposes there exists a program /bin/re1024, that reads a tape produced with an old 512 byte block *dump* resp. the new *dump.1b*, and writes it onto a 1024 byte filesystem.

BUGS

There is redundant information on the tape that could be used in case of tape reading problems. Unfortunately, *restor* doesn't use it.

NAME

rje – RJE (Remote Job Entry) to IBM

SYNOPSIS

/usr/rje/rjeinit
/usr/rje/rjehalt

DESCRIPTION

RJE is the communal name for a collection of programs and a file organization that allows a CADMUS system, equipped with a KEDQS driver, and associated KEDQS hardware to communicate with IBM's Job Entry Subsystems by mimicking an IBM 360 remote multileaving work station.

Implementation.

RJE is initiated by the command ***rjeinit*** and is terminated gracefully by the command ***rjehalt***. While active, RJE runs in the background and requires no human supervision. It quietly transmits, to the IBM system, jobs that have been queued by the ***send***(1C) command, and operator requests that have been entered by the ***rjestat***(1C) command. It receives, from the IBM system, print and punch data sets and message output. It enters the data sets into the proper UNIX directory and notifies the appropriate user of their arrival. It scans the message output to maintain a record on each of its jobs. It also makes these messages available for public inspection, so that ***rjestat***(1C), in particular, may extract responses.

Unless otherwise specified, all files and commands described below reside in directory **/usr/rje** (first exceptions: ***send*** and ***rjestat***).

There are two sources of data to be transmitted by RJE from UNIX to an IBM System/370. In both cases, the data is organized as files in the **/usr/rje/squeue** directory. The first are files named **co*** which are created by the enquiry command ***rjestat***(1C). The second source, containing the bulk of the data, are files named **rd*** or **sq*** which have been created by ***send*** and queued by the program ***rjeqer***. On completion of processing ***send*** invokes ***rjeqer***. ***Rjeqer*** and ***rjestat*** inform the program ***rjexmit*** that a file has been queued via the file **joblog**. Upon successful transmission of the data to the IBM machine, ***rjexmit*** removes the queued file. As files are transmitted and received, the program ***rjedisp*** writes an entry containing the date, time, file name, logname, and number of records in the file **acctlog**, if it exists. This file can be used for local logging or accounting information, but is not used elsewhere by RJE. The use of this information is up to the RJE administrator.

Each time ***rjeinit*** is invoked, the **joblog** file is truncated and recreated from the contents of the **/usr/rje/squeue** directory. During this time, ***rjeinit*** prevents simultaneous updating of the **joblog** file.

Output from the IBM system is classified as either a print data set, a punch data set, or message output. Print output is converted to an ASCII text file, with standard tabs. Form feeds are suppressed, but the last line of each page is distinguished by the presence of an extraneous trailing space. Punch output is not converted at all. This classification and the conversion occur as the output is received. Files are moved or copied into the appropriate user's directory and assigned the name **prnt*** or **pnch***, respectively, or placed into user directories under user-specified names, or used as input to programs to be automatically executed, as specified by the user. This process is driven by the "usr=..." specification. RJE retains ownership of these files and permits read-only access to them. Message output is digested by RJE immediately and is not retained.

A record is maintained for each job that passes through RJE. Identifying information is extracted contextually from files transmitted to and received from the IBM system. This information is stored and used by the *rjedisp* program for IBM job acknowledgements and delivery of output files.

The IBM system automatically returns an acknowledgement message for each job it receives. Other status messages are returned in response to enquiries entered by users. All messages received by RJE are appended to the **resp** file. The **resp** file is automatically truncated when it reaches 70,000 bytes.

While it is active, RJE occupies at least the three process slots that are appropriated by *rjeinit*. These slots are used to run *rjexmit*, the transmitter, *rjerecv*, the receiver, and *rjedisp*, the dispatcher. These three processes are connected by pipes. The function of each is as follows:

rjexmit
: Cycles repetitively, looking for data to transmit to the IBM system. After transmission, *rjexmit* passes an event notice to *rjedisp*. If *rjexmit* encounters a **stop** file, (created by *rjehalt*), it exits normally. In the case of error termination, *rjexmit* reboots RJE by executing *rjeinit*.

rjerecv
: Cycles repetitively, looking for data returning from the IBM machine. Upon receipt of data, *rjerecv* notifies either *rjexmit* or *rjedisp* of the event (transfer information is sometimes passed to *rjexmit*). *Rjerecv* exits normally at the first appropriate moment when it encounters the file **stop**, or exits reluctantly when it encounters a run of errors.

rjedisp
: Follows up event notices by directing output files, updating records, and notifying users. *Rjedisp* references the system files **/etc/passwd** and **/etc/utmp** to correlate user names, numeric ids, and terminals. Termination of *rjerecv* causes *rjedisp* to exit also.

Rjeinit has the capability of *dialing* any remote IBM system with the proper hardware and software configuration.

Most RJE files and directories are protected from unauthorized tampering. The exception is the **spool** directory. It is used by *send*(1C) to create temporary files in the correct file system. *Rjeqer* and *rjestat*(1C), the user's interfaces to RJE, operate in *setuid* mode to contribute the necessary permission modes.

Administration.
: Some minimal oversight of each RJE subsystem is required. The RJE mailbox should be inspected and cleaned out periodically. The **job** directory

should also be checked. The only files placed there are output files whose destination file systems are out of space. Users should be given a short period of time (say, a day or two), and then these files should be removed.

The configuration table **/usr/rje/lines** is accessed by all components of RJE. Each line of the table (maximum of 8) defines an RJE connection. Its seven columns may be labeled ***host***, ***system***, ***directory***, ***prefix***, ***device***, *peripherals* and *parameters*. These columns are described as follows:

host
: The name of a remote IBM computer (e.g., **A B C**). This string can be up to 5 characters.

system
: The name of a UNIX system. This name should be the same as the system name from *uname*(1).

directory
: This is the directory name of the servicing RJE subsystem (e.g., **/usr/rje1**).

prefix
: This is the string prefixed (redundantly) to several crucial files and programs in **directory** (e.g., **rje1**, **rje2**, **rje3**).

device
: This is the name of the controlling KEDQS device, with **/dev/** excised.

peripherals
: This field contains information on the logical devices (readers, printers, punches) used by RJE. Each subfield is separated by :. and is described as follows:

(1) Number of logical readers.
(2) Number of logical printers.
(3) Number of logical punches.

Note: the number of peripherals specified for an RJE subsystem **must** agree with the number of peripherals which have been described on the remote machine for that line.

parameters
: This field contains information on the type of connection to make. Each subfield is separated by :. Any or all fields may be omitted; however, the fields are positional. All but trailing delimiters must be present. For example, in

```
1200:512:::9-555-1212
```

subfields 3 and 4 are missing, but the delimiters are present. Each subfield is defined as follows:

(1) **space**
: This subfield specifies the amount of space (*S*) in blocks that RJE tries to maintain on file systems it touches. The default is 0 blocks. *Send* will not submit jobs and *rjeinit* issues a warning when less than 1.5*S* blocks are available;

rjerecv stops accepting output from the host when the capacity falls to *S* blocks; RJE becomes dormant, until conditions improve. If the space on the file system specified by the user on the "usr=" card would be depleted to a point below *S*, the file will be put in the **job** subdirectory of the connection's home directory, rather than in the place that the user requested.

(2) **size**
This subfield specifies the size in blocks of the largest file that can be accepted from the host without truncation taking place. The default is no truncation.

(3) **badjobs**
This subfield specifies what to do with undeliverable returning jobs. If an output file is undeliverable for any reason other than file system space limitations (e.g., missing or invalid "usr=" card) and this subfield contains the letter **y**, the output will be retained in the **job** subdirectory of the home directory, and login **rje** is notified. If this subfield contains an **n** or has any other value, undeliverable output will be discarded. The default is **n**.

(4) **console**
This subfield specifies the status of the interactive status terminal for this line. If the subfield contains an **i**, all console status facilities are inhibited (e.g., *rjestat*(1C) will not behave like a status terminal). In all cases, the normal non-interactive uses of *rjestat*(1C) will continue to function. The default is **y**.

(5) **dial-up**
This subfield contains a telephone number to be used to call a host machine. The telephone number may contain the digits 0 thru 9 and the character – which denotes a pause. If the telephone number is not present, no dialing is attempted and a leased line is assumed.

Sign-on is controlled by the existence of a **signon** file in the home directory. If this file is present, its contents are sent as a sign-on message to the host system. If this file does not exist, a blank card is sent. Sign-off is controlled in the same way, except that the **signoff** file is sent by *rjehalt* if it exists. If the **signoff** file does not exist, a "**/*signoff**" card is sent. These files should be ASCII text and no more than 80 characters.

Send(1C) and *rjestat*(1C) select an available connection by indexing on the **host** field of the configuration table. RJE programs index on the **prefix** field. A subordinate directory, **sque**, exists in **/usr/rje** for use by *rjedisp* and *shqer* programs. This directory holds those output files that have been designated as standard input to some executable file. This designation is done via the "usr=..." specification. *Rjedisp* places the output files here and updates the file **log** to specify the order of execution, arguments to be passed, etc. *Shqer* executes the appropriate files.

All RJE programs are shared text; therefore, if more than one RJE is to be run on a given UNIX system, simply link (via *ln*(1)) RJE2 program names to

RJE names in */usr*.

SEE ALSO

rjestat(1C), send(1C).,
UNIX Remote Job Entry User's Guide by K. A. Kelleman.
UNIX Remote Job Entry Administrative Guide by M. J. Fitton.
Setting Up UNIX.

DIAGNOSTICS

Rjeinit provides brief error messages describing obstacles encountered while bringing up RJE. They can best be understood in the context of the RJE source code. The most frequently occurring one is "cannot open /dev/vpm?". This may occur if another process already has the KEDQS device open.

Once RJE has been started, users should assist in monitoring its performance, and should notify operations personnel of any perceived need for remedial action. *Rjestat*(1C) will aid in diagnosing the current state of RJE. It can detect, with some reliability, when the far end of the communications line has gone dead, and will report in this case that the host computer is not responding to RJE. It will also attempt to reboot RJE if it detects a prolonged period of inactivity on the KEDQS.

NAME

rldown – power down sequence for RL02 disks (WDC11 emulation)

SYNOPSIS

/etc/rldown

DESCRIPTION

The CADMUS 9212 system uses the ANDROMEDA WDC11 controller to emulate RL02 disks. This controller provides a special command to ensure that the heads step to the center of the drives and are pulled into the head protection latches. Performing this command on other controllers causes a bus error and system crash.

Rldown must be the last command before switching the power off. Otherwise you risk bad blocks on your disk. First you will be asked if you are aware of what you are doing. After confirmation it starts a sync, waits some seconds until all I/O has finished and executes this special command. When this command is finished, you get a message on the console and the system hangs. Only reset or power off/on puts you back in control.

Rldown must be executed in **single user mode**! Any pending I/O can cause a damage of involved filesystems.

Therefore login as root at the console and type...

```
# init s
#
INIT: New run level: S
INIT: SINGLE USER MODE
# rldown
```

or...

```
# shutdown -h +1
Shutdown at ...
# rldown
```

Switch the power off.

BUGS

Problems arise if you use rldown in multi user mode: Only the I/O queue for the RL02 disks is checked before the heads are stepped to the center of the disks and the system is shut down.

NAME

/etc/rmsys – remove a remote system name

SYNOPSIS

/etc/rmsys name

DESCRIPTION

This program removes the special Newcastle Connection entry given by name.

FILES

/etc/utab - file of remote system names and identifiers

SEE ALSO

mksys(8N), utab(5N)

NAME

runacct – run daily accounting

SYNOPSIS

/usr/lib/acct/runacct [mmdd [state]]

DESCRIPTION

Runacct is the main daily accounting shell procedure. It is normally initiated via *cron*(8). *Runacct* processes connect, fee, disk, and process accounting files. It also prepares summary files for *prdaily* or billing purposes.

Runacct takes care not to damage active accounting files or summary files in the event of errors. It records its progress by writing descriptive diagnostic messages into **active**. When an error is detected, a message is written to **/dev/console**, mail (see *mail*(1)) is sent to **root** and **adm**, and *runacct* terminates. *Runacct* uses a series of lock files to protect against re-invocation. The files **lock** and **lock1** are used to prevent simultaneous invocation, and **lastdate** is used to prevent more than one invocation per day.

Runacct breaks its processing into separate, restartable *states* using **statefile** to remember the last *state* completed. It accomplishes this by writing the *state* name into **statefile**. *Runacct* then looks in **statefile** to see what it has done and to determine what to process next. *States* are executed in the following order:

SETUP	Move active accounting files into working files.
WTMPFIX	Verify integrity of **wtmp** file, correcting date changes if necessary.
CONNECT1	Produce connect session records in **ctmp.h** format.
CONNECT2	Convert **ctmp.h** records into **tacct.h** format.
PROCESS	Convert process accounting records into **tacct.h** format.
MERGE	Merge the connect and process accounting records.
FEES	Convert output of *chargefee* into **tacct.h** format and merge with connect and process accounting records.
DISK	Merge disk accounting records with connect, process, and fee accounting records.
MERGETACCT	Merge the daily total accounting records in **daytacct** with the summary total accounting records in **/usr/adm/acct/sum/tacct**.
CMS	Produce command summaries.
USEREXIT	Any installation-dependent accounting programs can be included here.

CLEANUP
Cleanup temporary files and exit.

To restart *runacct* after a failure, first check the **active** file for diagnostics, then fix up any corrupted data files such as **pacct** or **wtmp**. The lock files and **lastdate** file must be removed before *runacct* can be restarted. The argument *mmdd* is necessary if *runacct* is being restarted, and specifies the month and day for which *runacct* will rerun the accounting. Entry point for processing is based on the contents of **statefile**; to override this, include the desired *state* on the command line to designate where processing should begin.

EXAMPLES

To start *runacct*.

```
nohup runacct 2> /usr/adm/acct/nite/fd2log &
```

To restart *runacct*.

```
nohup runacct 0601 2>> /usr/adm/acct/nite/fd2log &
```

To restart *runacct* at a specific *state*.

```
nohup runacct 0601 MERGE 2>> /usr/adm/acct/nite/fd2log &
```

FILES

/etc/wtmp
/usr/adm/pacct*
/usr/include/tacct.h
/usr/include/ctmp.h
/usr/adm/acct/nite/active
/usr/adm/acct/nite/daytacct
/usr/adm/acct/nite/lock
/usr/adm/acct/nite/lock1
/usr/adm/acct/nite/lastdate
/usr/adm/acct/nite/statefile
/usr/adm/acct/nite/ptacct*.*mmdd*

SEE ALSO

acct(8), acctcms(8), acctcom(1), acctcon(8), acctmerg(8), acctprc(8), acctsh(8), cron(8), fwtmp(8), acct(2), acct(5), utmp(5).
UNIX Accounting System

DIAGNOSTICS

The accounting system will start complaining with *****RECOMPILE pnpsplit WITH NEW HOLIDAYS***** after the last holiday of the year. See *The UNIX System Accounting* for more on how to correct this condition. Other diagnostics are placed in various error and log files.

BUGS

Normally it is not a good idea to restart *runacct* in the **SETUP** *state*. Run **SETUP** manually and restart via:

runacct *mmdd* **WTMPFIX**

If *runacct* failed in the **PROCESS** *state*, remove the last **ptacct** file because it will not be complete.

NAME

rxtest – test a floppy disk for bad blocks

SYNOPSIS

rxtest [-w] device [blocks]

DESCRIPTION

Rxtest is a floppy disk check program similar to the standalone disk check program *check*. *Rxtest* tests floppies for the location of bad blocks but does not write any bad block information on the floppy. If there is an error in a header, or if there is a read, write or compare error within one block that block is defined as *bad*.
There is no way to replace bad blocks by good blocks except to use another floppy. *Rxtest* is only provided as a tool for separating a set of floppies into good ones and bad ones.

Device is the character special file name of a floppy drive such as */dev/rrx2* for example. The *blocks* argument gives the number of 512-byte blocks to be tested. The default value is *2002*, fitting to a double density, double sided 8 inch floppy. See the table in *rx(4)* for other floppy types. For the 5 1/4 inch floppy we recommend to specify only *980* blocks as the controller may use the remaining blocks for alignment information.
With the *–w* flag present the contents of the floppy is overwritten. Otherwise it is preserved during the test.

The teststrategy is as follows:

> A fixed number of blocks is read to save the contents, then written with a testpattern. That number of blocks is read again and compared with the testpattern. Afterwards the previous contents is restored by a write operation.
> The pattern *00ff,02fd,...fc,03fe,01* is used twice to test a 512-byte block. If the *–w* flag is specified the complete floppy is written first and then read and compared. Saving and restoring of the contents is omitted.

Testing a 5 1/4 inch floppy without preserving the contents takes about 2:20 minutes.

EXAMPLES

```
rxtest -w /dev/rrx2 980
rxtest /dev/rrx3
```

DIAGNOSTICS

Rxtest tells about the number of blocks tested, the number of bad blocks found and whether a block was indicated as bad by a read, write or compare error.
If you start *rxtest* from the system-console you will get intermixed diagnostics from the floppy driver and from *rxtest*.

SEE ALSO

rxctrl(1), rx(4), format(8)

BUGS

The number of bad blocks reported is just the sum of read, write and compare errors detected. So sometimes a block with write and read error is counted twice.

NAME

sa, accton – system accounting

SYNOPSIS

/etc/sa [**–abcdDfijkKlnrstuv**] [file]

/etc/accton [file]

DESCRIPTION

With an argument naming an existing *file*, *accton* causes system accounting information for every process executed to be placed at the end of the file. If no argument is given, accounting is turned off.

Sa reports on, cleans up, and generally maintains accounting files.

Sa is able to condense the information in */usr/adm/acct* into a summary file */usr/adm/savacct* which contains a count of the number of times each command was called and the time resources consumed. This condensation is desirable because on a large system */usr/adm/acct* can grow by 100 blocks per day. The summary file is normally read before the accounting file, so the reports include all available information.

If a file name is given as the last argument, that file will be treated as the accounting file; */usr/adm/acct* is the default.

Output fields are labelled: cpu for the sum of user+system time (in minutes), re for real time (also in minutes), k for cpu-time averaged core usage (in 1k units), avio for average number of i/o operations per execution. With options fields labelled tio for total i/o operations, k*sec for cpu storage integral (kilo-core seconds), u and s for user and system cpu time alone (both in minutes) will sometimes appear.

There are near a googol of options:

a	Place all command names containing unprintable characters and those used only once under the name '***other.'
b	Sort output by sum of user and system time divided by number of calls. Default sort is by sum of user and system times.
c	Besides total user, system, and real time for each command print percentage of total time over all commands.
d	Sort by average number of disk i/o operations.
D	Print and sort by total number of disk i/o operations.
f	Force no interactive threshold compression with –v flag.
i	Don't read in summary file.
j	Instead of total minutes time for each category, give seconds per call.
k	Sort by cpu-time average memory usage.
K	Print and sort by cpu-storage integral.
l	Separate system and user time; normally they are combined.
m	Print number of processes and number of CPU minutes for each user.

n	Sort by number of calls.
r	Reverse order of sort.
s	Merge accounting file into summary file */usr/adm/savacct* when done.
t	For each command report ratio of real time to the sum of user and system times.
u	Superseding all other flags, print for each command in the accounting file the user ID and command name.
v	Followed by a number *n*, types the name of each command used *n* times or fewer. Await a reply from the terminal; if it begins with 'y', add the command to the category '**junk**.' This is used to strip out garbage.

FILES

/usr/adm/acct	raw accounting
/usr/adm/savacct	summary
/usr/adm/usracct	per-user summary

SEE ALSO

acct(2)

BUGS

The number of options to this program is absurd.

NAME

sa1, sa2, sadc – system activity report package

SYNOPSIS

/usr/lib/sa/sadc [t n] [ofile]

/usr/lib/sa/sa1 [t n]

/usr/lib/sa/sa2 [**–ubdycwaqvm**] [**–s** time] [**–e** time] [**–i** sec]

DESCRIPTION

System activity data can be accessed at the special request of a user (see *sar*(1)) and automatically on a routine basis as described here. The operating system contains a number of counters that are incremented as various system actions occur. These include CPU utilization counters, buffer usage counters, disk and tape I/O activity counters, TTY device activity counters, switching and system-call counters, file-access counters, queue activity counters, and counters for inter-process communications.

Sadc and shell procedures *sa1* and *sa2* are used to sample, save and process this data.

Sadc, the data collector, samples system data *n* times every *t* seconds and writes in binary format to *ofile* or to standard output. If *t* and *n* are omitted, a special record is written. This facility is used at system boot time to mark the time at which the counters restart from zero. The */etc/rc* entry:

```
        su sys –c "/usr/lib/sa/sadc /usr/adm/sa/sa`date +%d`&"
```

writes the special record to the daily data file to mark the system restart.

The shell script *sa1*, a variant of *sadc*, is used to collect and store data in binary file **/usr/adm/sa/sa***dd* where *dd* is the current day. The arguments *t* and *n* cause records to be written *n* times at an interval of *t* seconds, or once if omitted. The entries in **crontab** (see *cron*(8)):

```
        0 * * * 0,6 su sys –c "/usr/lib/sa/sa1"
        0 8–17 * * 1–5 su sys –c "/usr/lib/sa/sa1 1200 3"
        0 18–7 * * 1–5 su sys –c "/usr/lib/sa/sa1"
```

will produce records every 20 minutes during working hours and hourly otherwise.

The shell script *sa2*, a variant of *sar*(1), writes a daily report in file **/usr/adm/sa/sar***dd*. The options are explained in *sar*(1). The **crontab** entry:

```
        5 18 * * 1–5 su adm –c "/usr/lib/sa/sa2 –s 8:00 –e 18:01 –i 3600 –A"
```

will report important activities hourly during the working day.

The structure of the binary daily data file is:

```
struct sa {
        struct sysinfo si;  /* see /usr/include/sys/sysinfo.h */
        int  szinode;       /* current entries of inode table  */
        int  szfile;        /* current entries of file table  */
        int  sztext;        /* current entries of text table  */
        int  szproc;        /* current entries of proc table  */
        int  mszinode;      /* size of inode table  */
        int  mszfile;       /* size of file table  */
        int  msztext;       /* size of text table  */
        int  mszproc;       /* size of proc table  */
        long  inodeovf;     /* cumul. overflows of inode table  */
        long  inodeovf;     /* cumul. overflows of file table  */
        long  textovf;      /* cumul. overflows of text table  */
        long  procovf;      /* cumul. overflows of proc table  */
        time_t  ts;         /* time stamp, seconds  */
        long  devio[NDEVS][4];     /* device info for up to NDEVS units  */
 #define IO_OPS           0        /* cumul. I/O requests  */
 #define IO_BCNT          1        /* cumul. blocks transferred */
 #define IO_ACT           2        /* cumul. drive busy time in ticks  */
 #define IO_RESP          3        /* cumul. I/O resp time in ticks  */
 };
```

FILES

/usr/adm/sa/sa*dd*	daily data file
/usr/adm/sa/sar*dd*	daily report file
/tmp/sa.adrfl	address file

SEE ALSO
sag(1G), sar(1), timex(1).

NAME

sendnews – send news articles via mail

SYNOPSIS

sendnews [**–o**] [**–a**] [**–b**] [**–n** newsgroups] destination

DESCRIPTION

sendnews reads an article from it's standard input, performs a set of changes to it, and gives it to the mail program to mail it to *destination*.

An 'N' is prepended to each line for decoding by *uurec(8)*.

The **–o** flag handles old format articles.

The **–a** flag is used for sending articles via the **ARPANET**. It maps the article's path from *uucphost!xxx* to *xxx@arpahost*.

The **–b** flag is used for sending articles via the **Berknet**. It maps the article's path from *uucphost!xxx* to *berkhost:xxx*.

The **–n** flag changes the article's newsgroup to the specified *newsgroup*.

SEE ALSO

inews(1), uurec(8), recnews(8), readnews(1), checknews(1)

NAME

setmnt – establish mount table

SYNOPSIS

/etc/setmnt

DESCRIPTION

Setmnt creates the **/etc/mnttab** table (see *mnttab*(5)), which is needed for both the *mount*(8) and *umount* commands. *Setmnt* reads standard input and creates a *mnttab* entry for each line. Input lines have the format:

filesys node

where *filesys* is the name of the file system's *special file* (e.g., "rp??") and *node* is the root name of that file system. Thus *filesys* and *node* become the first two strings in the *mnttab*(5) entry.

FILES

/etc/mnttab

SEE ALSO

mnttab(5). devnm(8)

BUGS

Evil things will happen if *filesys* or *node* are longer than 10 characters. *Setmnt* silently enforces an upper limit on the maximum number of *mnttab* entries.

NAME

/etc/setugi – alter user id of a UNIX server

SYNOPSIS

/etc/setugi

DESCRIPTION

This program is invoked by a UNIX server when its client process carries out a set user/group id execution. The effect of executing it is to alter the user/group id of the server. The program must be setuid to "root".

SEE ALSO

usrv(8N)

NAME

shutdown – terminate all processing

SYNOPSIS

/etc/shutdown.sh

DESCRIPTION

Shutdown is part of the UNIX operation procedures. Its primary function is to terminate all currently running processes in an orderly and cautious manner. The procedure is designed to interact with the operator (i.e., the person who invoked *shutdown*). *Shutdown* may instruct the operator to perform some specific tasks, or to supply certain responses before execution can resume. *Shutdown* goes through the following steps:

- All users logged on the system are notified to log off the system by a broadcasted message. The operator may display his/her own message at this time. Otherwise, the standard file save message is displayed.
- If the operator wishes to run the file-save procedure, *shutdown* unmounts all file systems.
- All file systems' super blocks are updated before the system is to be stopped (see *sync*(8)). This must be done before re-booting the system, to insure file system integrity.

Berkeley shutdown

The program /etc/shutdown from Berkeley is best used as */etc/shutdown -h +n*, where n is the number of minutes when the system shall shut down. The fastest way to shut the system down with notification of users is */etc/shutdown -h +1* , without notification, */etc/init s*. After shutdown, you will be in single user mode. Type *sync* and press the INIT-button. Power off.

DIAGNOSTICS

The most common error diagnostic that will occur is *device busy*. This diagnostic happens when a particular file system could not be unmounted. See *umount*(8).

NAME

/etc/startnc, /etc/stopnc – starts up (closes down) the file server spawner

SYNOPSIS

/etc/startnc [-d]
/etc/stopnc

DESCRIPTION

/etc/startnc starts up the UNIX server spawner, in the file "/etc/usam", and stores its process id in the file "/etc/usampid". If the spawner was already running, the program will shut it down before starting the new process. The program also handles new releases of the server software, which should be placed in the files "/etc/usrv.new", "/etc/setugi.new" and "/etc/usam.new" (the Newcastle Connection make files will do this automatically). The versions being replaced will be moved to the files "/etc/usrv.old", "/etc/setugi.old" and "/etc/usam.old". The option **-d** is used to select the version of the spawner held in the file "/etc/usam.dbg". This is conventionally a debugging version of the spawner, and the option should only be used whilst testing the system.

/etc/stopnc closes down the UNIX server spawner by sending the signal SIGTERM to the process whose id is contained in the file "/etc/usampid".

FILES

/etc/usampid
/etc/usam
/etc/usam.dbg
/etc/usam.new
/etc/setugi
/etc/setugi.new
/etc/setugi.old
/etc/usrv.new
/etc/usrv.old
/etc/usrv.old

SEE ALSO

usrv(8N), setugi(8N), usam(8N)

DIAGNOSTICS

A message will be printed when there is a new software release installed. If the caller is not superuser or if there is a problem with the execution of the spawner program, an error message will be printed.

NAME

sync – update the super block

SYNOPSIS

sync

DESCRIPTION

Sync executes the *sync* system primitive. If the system is to be stopped, *sync* must be called to insure file system integrity. See *sync*(2) for details.

SEE ALSO

sync(2)

NAME

unite – enable a remote user to access the local system

SYNOPSIS

unite [-dgprv] [system [remote_id [local_id]]]

DESCRIPTION

Establishes *local_id* as the local surrogate for user *remote_id* on remote Unix *system*. e.g.

```
unite /../unix4 dave david
```

lets user 'dave' on system '/../unix4' execute processes on the local machine, as if he had logged in as 'david'. A missing local_id is assumed to have the same name as the remote user. A missing remote_id is assumed to mean each user of the remote system is to be mapped to the local user of the same name. Unite normally refuses to map "root" to "root"; any user names with userid 0 are ignored. *Unite* without parameters prints out the current list of remote and local user pairs.

Option "-d" deletes the remote system or user named.

Option "-g" applies *unite* to groups instead of users.

Option "-p" can be used to print a single pair, or the entries for a single system.

Option "-r" overrides the default control which will not normally map "root" (or any user name with userid 0) to local "root". It is only effective when an entire system is being united, and has no meaning if combined with '-g' option.

Option "-v" announces each item as it is created.

Both remote_id and local_id may be in numeric form. In this case, the "/etc/passwd" ("/etc/group") file on the relevant system is not accessed. This is useful when creating a United system.

FILES

/etc/pwmap /etc/groupmap, user and group mappings;
system/etc/passwd
system/etc/group
/etc/passwd /etc/group - local and remote user and
        group tables.
/etc/utab - table of systems.

DIAGNOSTICS

Complains about incorrect parameters such as non-existent ids or systems.

SEE ALSO

pwmap(5N), utab(5N)

NAME

/etc/usam – initiate a UNIX server for a remote client

SYNOPSIS

/etc/usam [root directory [working directory]]

DESCRIPTION

This program listens on a fixed port number for an incoming request for remote service. In response, it initiates a UNIX server on another port and returns this port number to the client, who now deals directly with its own UNIX server. The program also performs user/group validation and mapping for the incoming request, allowing the local system manager to maintain control of the user population. The parameters passed to the spawner allow the initiator to control exactly where the spawner lives in the file store hierarchy, and therefore to control where incoming users' UNIX servers live and the image of the file system that those users see. The default value for both fields is "/".

The fixed port number used by all spawners on your network is controlled by the macros "SET_USAM_PORT" and "USAM_INIT" in the file "h/netlocal.h" of the distribution directory of the Newcastle Connection.

FILES

/etc/pwmap, /etc/groupmap

SEE ALSO

usrv(8N), unite(8N), startnc(8N), stopnc(8N), pwmap(5N), utab(5N), "The Newcastle Connection – Release 1.0: Network Interface Installation Guide"

DIAGNOSTICS

Reports will be given on the console in the event of errors.

NAME

usrv – UNIX server for a remote client

SYNOPSIS

/etc/usrv

DESCRIPTION

This program is spawned in response to incoming requests for service and provides a remote user with the facilities of the normal UNIX system file interface.

SEE ALSO

usam(8N)

DIAGNOSTICS

Standard UNIX error return codes are handed back to clients in the external "errno" of the caller's program.

NAME

uuclean – uucp spool directory clean-up

SYNOPSIS

/usr/lib/uucp/uuclean [options]

DESCRIPTION

Uuclean will scan the spool directory for files with the specified prefix and delete all those which are older than the specified number of hours.

The following options are available.

–d*directory*	Clean *directory* instead of the spool directory.
–p*pre*	Scan for files with *pre* as the file prefix. Up to 10 **–p** arguments may be specified. A **–p** without any *pre* following will cause all files older than the specified time to be deleted.
–n*time*	Files whose age is more than *time* hours will be deleted if the prefix test is satisfied. (default time is 72 hours)
–w*file*	The default action for *uuclean* is to remove files which are older than a specified time (see **–n** option). The **–w** option is used to find those files older than *time* hours, however, the files are not deleted. If the argument *file* is present the warning is placed in *file*, otherwise, the warnings will go to the standard output.
–s*sys*	Only files destined for system *sys* are examined. Up to 10 **–s** arguments may be specified.
–m*file*	The **–m** option sends mail to the owner of the file when it is deleted. If a *file* is specified then an entry is placed in *file*.

This program is typically started by *cron*(8).

FILES

/usr/lib/uucp
    directory with commands used by *uuclean* internally
/usr/spool/uucp
    spool directory

SEE ALSO

cron(8), uucp(1C), uux(1C).

NAME

uucp – uucp installation made easy

DESCRIPTION

Consider the simple case of connecting two systems over a direct permanent line, i.e. no modems. The names of the two systems are alpha and beta. Alphas terminal is called /dev/ttya, betas terminal /dev/ttyb.

An uucp link is asymmetric in nature: one port sends a login message to the other port. The first port must not have a shell enabled on the line, the second port must. Let us assume alpha calls beta. So in the file /etc/inittab for alpha there is no line for ttya, or the entry has a number different from 2 in the run level field. On beta there must be an entry in /etc/inittab for ttyb with run level 2. So, if terminals were attached to ttya and ttyb, you could login at ttyb, but not on ttya. Then connect ttya and ttyb with a cable that switches TxDATA and RxDATA (pins 2 and 3 on the Canon RS232 connector). Login on alpha on any terminal as root and execute "cu -t -a /dev/null -l /dev/ttya". You must now be able to login normally on beta. Thus, we proved that the hardware link is ok.

On each system login as root and give nuucp a password. E.g. on alpha you enter "passwd nuucp" and then as password thisisalpha, on beta the password thisisbeta.

Next, establish in alphas and betas /usr/lib/uucp directory the files L.sys and L-devices. Our L.sys format is an extension of the standard; read the file /usr/lib/uucp/L.sys.format for a description. Alphas L.sys will look like this:

```
beta Any ttya 9600 ttya \r ?login-\r-?login nuucp\r ?ssword:
thisisbeta\r
```

Alphas L-devices will look like this:

```
DIR ttya 0 9600
```

Betas L.sys will look like this:

```
alpha None ttyb 9600 ttyb \r ?login-\r-?login nuucp\r ?ssword:
thisisalpha\r
```

and its L-devices:

```
DIR ttyb 0 9600
```

Note the difference between None and Any: alpha can call at "Any" time, beta can call at "None" time (i.e. never). Make sure that L.sys and L-devices have owner uucp, L.sys should have mode 0400.

On alpha, copy a small file to /tmp and enter "uucp -r /tmp/file beta!/tmp". The -r option prevents uucp from starting uucico, the actual transfer program. Go to /usr/spool/uucp and convince yourself that two files C.* and D.* have been created. Call uucico by hand with a debug option: "/usr/lib/uucp/uucico -r1 -sbeta -x6". Watch the messages and see how uucico tries to login at the other system. After a while both systems agree that no more files are to be exchanged (message H'Y') and uucico terminates.

Now try "mail beta!root" and type a small text, followed by CTRL-Z. This time uucico is called automatically. After a few seconds the mail on beta will have arrived.

Alpha should now poll beta regularly, so that when beta sends something to alpha, the delay will not exceed say half an hour. An entry for this has been prepared in your /usr/lib/crontab file.

Possible causes of error are: missing read and write permissions for ttya and ttyb; wrong ownership and mode for /usr/spool/uucp and /usr/lib/uucp. The following is ok:

```
drwxr-xr-x    uucp   /usr/lib/uucp
drwxrwxrwx    uucp   /usr/spool/uucp
-r--------    uucp   /usr/lib/uucp/L.sys
-r-sr-xr-x    uucp   /usr/lib/uucp/uucico
-r-sr-xr-x    uucp   /usr/lib/uucp/uuclean
-r-sr-xr-x    uucp   /usr/lib/uucp/uuxqt
```

If you have a modem, but no autodialler, then L.sys must also contain the "None" entry. So that you can be dialled, you will have to enable a login on the modem port. When you want to dial out, you must first disable the port. You edit /etc/inittab and change the run level 2 to a 0. Then you give the command "/etc/init q". This will kill the getty process for this line. Then you execute uucico with the -t option. This overrides the "None" entry in L.sys. A sample shell script for you is provided in /usr/local/calluucp.

Lets assume you say "uucp /usr/jim/filea beta!/usr/joe/fileb". On your machine the directories /, /usr and /usr/jim must have execute permission for others, and /usr/jim/filea must have read permission for others, so that uucico can access the file. On beta, again /, /usr and /usr/joe must have execute permission for others. If fileb exists, it must have write permission. If it does not exist, then directory /usr/joe must have write permission! If you think this is annoying, use uupick and uuput to transfer files.

NAME

uurec – receive processed news articles via mail

SYNOPSIS

uurec

DESCRIPTION

uurec reads news articles on the standard input sent by *sendnews(8)*, decodes them, and gives them to *inews(1)* for insertion.

SEE ALSO

inews(1), readnews(1), recnews(8), sendnews(8), checknews(1)

NAME

uusub − monitor uucp network

SYNOPSIS

/usr/lib/uucp/uusub [options]

DESCRIPTION

Uusub defines a *uucp* subnetwork and monitors the connection and traffic among the members of the subnetwork. The following options are available:

−a*sys*	Add *sys* to the subnetwork.
−d*sys*	Delete *sys* from the subnetwork.
−l	Report the statistics on connections.
−r	Report the statistics on traffic amount.
−f	Flush the connection statistics.
−u*hr*	Gather the traffic statistics over the past *hr* hours.
−c*sys*	Exercise the connection to the system *sys*. If *sys* is specified as **all**, then exercise the connection to all the systems in the subnetwork.

The meanings of the connections report are:

sys #call #ok time #dev #login #nack #other

where *sys* is the remote system name, *#call* is the number of times the local system tries to call *sys* since the last flush was done, *#ok* is the number of successful connections, *time* is the latest successful connect time, *#dev* is the number of unsuccessful connections because of no available device (e.g. ACU), *#login* is the number of unsuccessful connections because of login failure, *#nack* is the number of unsuccessful connections because of no response (e.g. line busy, system down), and *#other* is the number of unsuccessful connections because of other reasons.

The meanings of the traffic statistics are:

sfile sbyte rfile rbyte

where *sfile* is the number of files sent and *sbyte* is the number of bytes sent over the period of time indicated in the latest *uusub* command with the **−u***hr* option. Similarly, *rfile* and *rbyte* are the numbers of files and bytes received.

The command:

```
uusub −c all −u 24
```

is typically started by *cron*(8) once a day.

FILES

/usr/spool/uucp/SYSLOG	system log file
/usr/lib/uucp/L_sub	connection statistics
/usr/lib/uucp/R_sub	traffic statistics

SEE ALSO

uucp(1C), uustat(1C).

NAME

volcopy, labelit – copy file systems with label checking

SYNOPSIS

source target

/etc/volcopy [options] fsname special1 volname1 special2 volname2

~~**/etc/stvolcopy** [options] fsname special1 volname1 special2 volname2~~

/etc/labelit special [fsname volume [**–n**]]

DESCRIPTION

Volcopy makes a literal copy of the file system using a blocksize matched to the device. ~~The program *stvolcopy* is the same as *volcopy*, but is modified for the streamer to use larger block sizes and double buffered~~ I/O. *Options* are:

–a	invoke a verification sequence requiring a positive operator response instead of the standard 10 second delay before the copy is made,
–s	(default) invoke the **CTRL-C if wrong** verification sequence.

Other *options* are used only with tapes:

–bpidensity	bits-per-inch (i.e., **800/1600/6250**, or for 3B20S systems with Kennedy tape drives, **1600k**),
–feetsize	size of reel in feet (i.e., **1200/2400**),
–reelnum	beginning reel number for a restarted copy,
–buf	use double buffered I/O (not for stvolcopy).

The program requests length and density information if it is not given on the command line or is not recorded on an input tape label. If the file system is too large to fit on one reel, *volcopy* will prompt for additional reels. Labels of all reels are checked. Tapes may be mounted alternately on two or more drives.

1) The *fsname* argument represents the mounted name (e.g.: **root**, **u1**, etc.) of the filsystem being copied.

2) The *special* should be the physical disk section or tape (e.g.: **/dev/rhk2**, **/dev/rmt0**, etc.).

3) The *volname* is the physical volume name (e.g.: **pk3**, **t0122**, etc.) and should match the external label sticker. Such label names are limited to six or fewer characters. *Volname* may be – to use the existing volume name.

Special1 and *volname1* are the device and volume from which the copy of the file system is being extracted. *Special2* and *volname2* are the target device and volume.

Fsname and *volname* are recorded in the last 12 characters of the superblock (**char fsname[6], volname[6];**).

* *Labelit* can be used to provide initial labels for unmounted disk or tape file systems. With the optional arguments omitted, *labelit* prints current label values. The **–n** option provides for initial labeling of new tapes only (this destroys previous contents).

FILES

/etc/log/filesave.log	a record of file systems/volumes copied

SEE ALSO

fs(5).

BUGS

Only device names beginning **/dev/rmt**, **/dev/nrmt** or **/dev/rst**, **/dev/nrst** are treated as tapes. Tape record sizes are determined both by density and by drive type. On CADMUS systems, records are 5,120 bytes long at 800 and 1600 bits-per-inch, and 25,600 bytes long at 6250 bits-per-inch. The streamer is written with very large blocks, but a thing like record length does not really exist on streamers. A streamer can be read and written with any blocksize, this is only a matter of efficiency.

NAME

wall – write to all users

SYNOPSIS

/etc/wall

DESCRIPTION

Wall reads its standard input until an end-of-file. It then sends this message to all currently logged in users preceded by:

Broadcast Message from ...

It is used to warn all users, typically prior to shutting down the system.

The sender must be super-user to override any protections the users may have invoked (see *mesg*(1)).

FILES

/dev/tty*

SEE ALSO

mesg(1), write(1).

DIAGNOSTICS

"Cannot send to ..." when the open on a user's tty file fails.

NAME
whatconf – what device drivers are in an unix kernel

SYNOPSIS
/etc/whatconf unixkernel

DESCRIPTION
Whatconf tells you what devices the specified unix kernel is configured for.

EXAMPLE

```
/etc/whatconf /unix
```

SEE ALSO
newconf(8)

NAME
whodo – who is doing what

SYNOPSIS
/etc/whodo

DESCRIPTION
Whodo produces merged, reformatted, and dated output from the *who*(1) and *ps*(1) commands.

SEE ALSO
ps(1), who(1), w(1)